JADE PROGRAMMER'S GUIDE

USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

last update: 4-September-2001. JADE 2.4

Authors: Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (*ex* CSELT *now* TILab)
Giovanni Rimassa (University of Parma)

JADE - Java Agent DEvelopment Framework is a framework to develop multi-agent systems in compliance with the FIPA specifications. JADE successfully passed the 1st FIPA interoperability test in Seoul (Jan. 99) and the 2nd FIPA interoperability test in London (Apr. 01).

TABLE OF CONTENTS

1 INTRODUCTION

This programmer's guide is complemented by the administrator's guide and the HTML documentation available in the directory jade/doc. If and where conflict arises between what is reported in the HTML documentation and this guide, preference should be given to the HTML documentation that is updated more frequently.

JADE (Java Agent Development Framework) is a software development framework aimed at developing multi-agent systems and applications conforming to FIPA standards for intelligent agents. It includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. JADE has been fully coded in Java and an agent programmer, in order to exploit the framework, should code his/her agents in Java, following the implementation guidelines described in this programmer's guide.

This guide supposes the reader to be familiar with the FIPA standards[1], at least with the *Agent Management* specifications (FIPA no. 23), the *Agent Communication Language*, and the *ACL Message Structure* (FIPA no. 61).

JADE is written in Java language and is made of various Java packages, giving application programmers both ready-made pieces of functionality and abstract interfaces for custom, application dependent tasks. Java was the programming language of choice because of its many attractive features, particularly geared towards object-oriented programming in distributed heterogeneous environments; some of these features are Object Serialization, Reflection API and Remote Method Invocation (RMI).

JADE is composed of the following main packages.

`jade.core` implements the kernel of the system. It owns the `Agent` class that must be extended by application programmers; besides, a `Behaviour` class hierarchy is contained in `jade.core.behaviours` sub-package. Behaviours implement the tasks, or intentions, of an agent. They are logical activity units that can be composed in various ways to achieve complex execution patterns and that can be concurrently executed. Application programmers define agent operations writing behaviours and agent execution paths interconnecting them.

The `jade.lang` package has a sub-package for every language used in JADE. In particular, a `jade.lang.acl` sub-package is provided to process Agent Communication Language according to FIPA standard specifications. `jade.lang.sl` contains the SL-0 codec[2], both the parser and the encoder.

The `jade.onto` package contains a set of classes to support user-defined ontologies. It has a subpackage `jade.onto.basic` containing a set of basic concepts (i.e. Action, TruePredicate, FalsePredicate, …) that are usually part of every ontology, and a `BasicOntology` that can be joined with user-defined ontologies.

The `jade.domain` package contains all those Java classes that represent the Agent Management entities defined by the FIPA standard, in particular the AMS and DF agents, that provide life-cycle, white and yellow page services. The subpackage `jade.domain.FIPAAgentManagement` contains the FIPA-Agent-Management Ontology and all the classes representing its concepts. The subpackage

[1] See http://www.fipa.org/

[2] refer to FIPA document no. 8 for the specifications of the SL content language.

`jade.domain.JADEAgentManagement` contains, instead, the JADE extensions for Agent-Management (e.g. for sniffing messages, controlling the life-cycle of agents, …), including the Ontology and all the classes representing its concepts. The subpackage jade.domain.introspection contains the concepts used for the domain of discourse between the JADE tools (e.g. the Sniffer and the Introspector) and the JADE kernel.

The `jade.gui` package contains a set of generic classes useful to create GUIs to display and edit Agent-Identifiers, Agent Descriptions, ACLMessages, …

The `jade.mtp` package contains a Java interface that every Message Transport Protocol should implement in order to be readily integrated with the JADE framework, and the implementation of a set of these protocols.

`jade.proto` is the package that contains classes to model standard interaction protocols (i.e. *fipa-request*, *fipa-query*, *fipa-contract-net* and soon others defined by FIPA), as well as classes to help application programmers to create protocols of their own.

The `fipa` package contains the IDL module defined by FIPA for IIOP-based message transport.

Finally, the *jade.wrapper* package provides wrappers of the JADE higher-level functionalities that allows the usage of JADE as a library, where external Java applications launch JADE agents and agent containers (see also section 3.8).

JADE comes bundled with some tools that simplify platform administration and application development. Each tool is contained in a separate sub-package of `jade.tools`. Currently, the following tools are available:

- *Remote Management Agent, RMA* for short, acting as a graphical console for platform management and control. A first instance of an RMA can be started with a command line option ("*-gui*") , but then more than one GUI can be activated. JADE maintains coherence among multiple RMAs by simply multicasting events to all of them. Moreover, the RMA console is able to start other JADE tools.
- The *Dummy Agent* is a monitoring and debugging tool, made of a graphical user interface and an underlying JADE agent. Using the GUI it is possible to compose ACL messages and send them to other agents; it is also possible to display the list of all the ACL messages sent or received, completed with timestamp information in order to allow agent conversation recording and rehearsal.
- The *Sniffer* is an agent that can intercept ACL messages while they are in flight, and displays them graphically using a notation similar to UML sequence diagrams. It is useful for debugging your agent societies by observing how they exchange ACL messages.
- The *IntrospectorAgent* is a very useful tool that allows to monitor the life cycle of an agent and its exchanged ACL messages.
- The *SocketProxyAgent* is a simple agent, acting as a bidirectional gateway between a JADE platform and an ordinary TCP/IP connection. ACL messages, travelling over JADE proprietary transport service, are converted to simple ASCII strings and sent over a socket connection. Viceversa, ACL messages can be tunnelled via this TCP/IP connection into the JADE platform. This agent is useful, e.g. to handle network firewalls or to provide platform interactions with Java applets within a web browser.
- The *DF GUI* is a complete graphical user interface that is used by the default Directory Facilitator (DF) of JADE and that can also be used by every other DF that the user might need. In such a way, the user might create a complex network of domains and sub-domains of yellow pages. This GUI allows in a simple and intuitive

way to control the knowledge base of a DF, to federate a DF with other DF's, and to remotely control (register/deregister/modify/search) the knowledge base of the parent DF's and also the children DF's (implementing the network of domains and sub-domains).

JADE™ is a trade mark registered by CSELT[3].

2 JADE FEATURES

The following is the list of features that JADE offers to the agent programmer:

- Distributed agent platform. The agent platform can be split among several hosts (provided they can be connected via RMI). Only one Java application, and therefore only one Java Virtual Machine, is executed on each host. Agents are implemented as Java threads and live within *Agent Containers* that provide the runtime support to the agent execution.
- Graphical user interface to manage several agents and agent containers from a remote host.
- Debugging tools to help in developing multi agents applications based on JADE.
- Intra-platform agent mobility, including state and code of the agent.
- Support to the execution of multiple, parallel and concurrent agent activities via the behaviour model. JADE schedules the agent behaviours in a non-preemptive fashion.
- FIPA-compliant Agent Platform, which includes the *AMS* (*Agent Management System*), the *DF* (*Directory Facilitator*), and the *ACC* (*Agent Communication Channel*). All these three components are automatically activated at the agent platform start-up.
- Many FIPA-compliant DFs can be started at run time in order to implement multi-domain applications, where a domain is a logical set of agents, whose services are advertised through a common facilitator. Each DF inherits a GUI and all the standard capabilities defined by FIPA (i.e. capability of registering, deregistering, modifying and searching for agent descriptions; and capability of federating within a network of DF's).
- Efficient transport of ACL messages inside the same agent platform. Infact, messages are transferred encoded as Java objects, rather than strings, in order to avoid marshalling and unmarshalling procedures. When crossing platform boundaries, the message is automatically converted to/from the FIPA compliant syntax, encoding, and transport protocol. This conversion is transparent to the agent implementers that only need to deal with Java objects.
- Library of FIPA interaction protocols ready to be used.
- Automatic registration and deregistration of agents with the AMS.
- FIPA-compliant naming service: at start-up agents obtain their GUID (Globally Unique Identifier) from the platform.
- Support for application-defined content languages and ontologies.
- InProcess Interface to allow external applications to launch autonomous agents.

3 CREATING MULTI-AGENT SYSTEMS WITH JADE

This chapter describes the JADE classes that support the development of multi-agent systems. JADE warrants syntactical compliance and, where possible, semantic compliance with FIPA specifications.

[3] Since March 2001, the name of the company is changed into TILab.

3.1 The Agent Platform

The standard model of an agent platform, as defined by FIPA, is represented in the following figure.

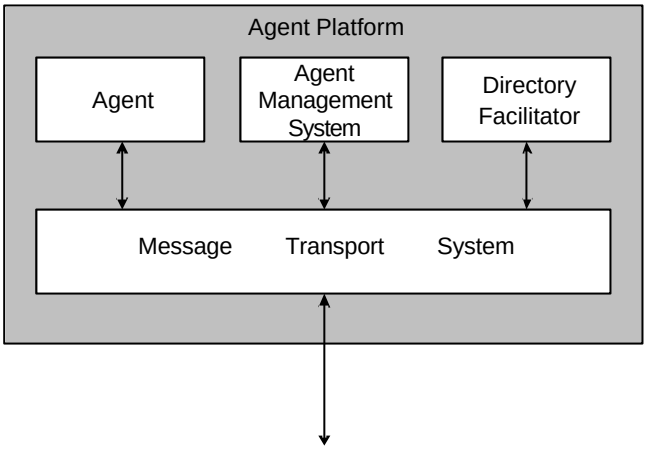

Figure 1 - Reference architecture of a FIPA Agent Platform

The Agent Management System (AMS) is the agent who exerts supervisory control over access to and use of the Agent Platform. Only one AMS will exist in a single platform. The AMS provides white-page and life-cycle service, maintaining a directory of agent identifiers (AID) and agent state. Each agent must register with an AMS in order to get a valid AID.

The Directory Facilitator (DF) is the agent who provides the default yellow page service in the platform.

The Message Transport System, also called Agent Communication Channel (ACC), is the software component controlling all the exchange of messages within the platform, including messages to/from remote platforms.

JADE fully complies with this reference architecture and when a JADE platform is launched, the AMS and DF are immediately created and the ACC module is set to allow message communication. The agent platform can be split on several hosts. Only one Java application, and therefore only one Java Virtual Machine (JVM), is executed on each host. Each JVM is a basic container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host. The main-container, or front-end, is the agent container where the AMS and DF lives and where the RMI registry, that is used internally by JADE, is created. The other agent containers, instead, connect to the main container and provide a complete run-time environment for the execution of any set of JADE agents.

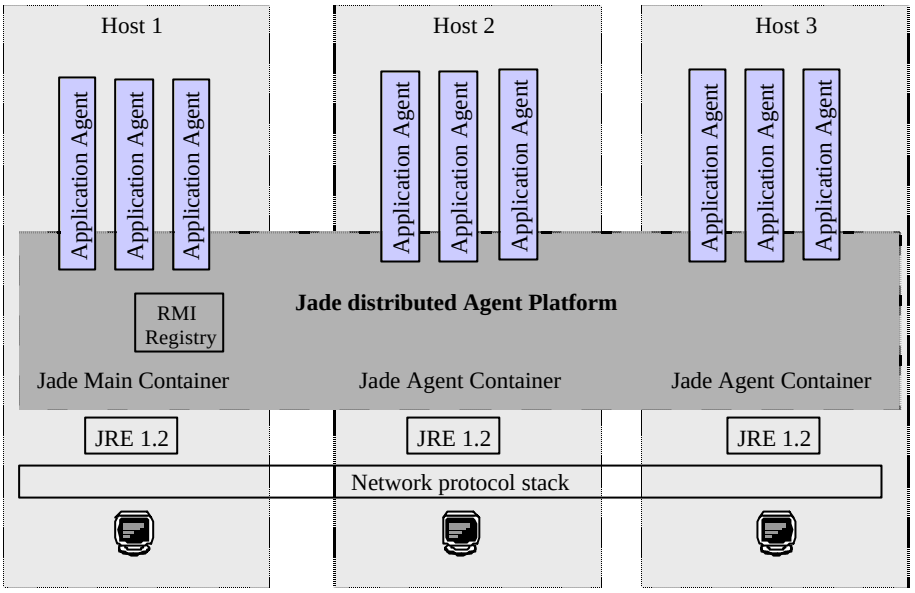

Figure 2 - JADE Agent Platform distributed over several containers

According to the FIPA specifications, DF and AMS agents communicate by using the `FIPA-SL0` content language, the `fipa-agent-management` ontology, and the `fipa-request` interaction protocol. JADE provides compliant implementations for all these components:

- the SL-0 content language is implemented by the class `jade.lang.sl.SL0Codec`. Automatic capability of using this language can be added to any agent by using the method `Agent.registerLanguage(SL0Codec.NAME, new SL0Codec());`
- concepts of the ontology (apart from Agent Identifier, implemented by `jade.core.AID`) are implemented by classes in the `jade.domain.FIPAAgentManagement` package. The `FIPAAgentManagementOntology` class defines the vocabulary with all the constant symbols of the ontology. Automatic capability of using this ontology can be added to any agent by using the fillowing code:
  `Agent.registerOntology(FIPAAgentManagementOntology.NAME, FIPAAgentManagementOntology.instance());`
- finally, the `fipa-request` interaction protocol is implemented as ready-to-use behaviours in the package `jade.proto`.

3.1.1 FIPA-Agent-Management ontology

Every class implementing a concept of the `fipa-agent-management` ontology is a simple collection of attributes, with public methods to read and write them, according to the frame based model that represents FIPA `fipa-agent-management` ontology concepts. The following convention has been used. For each attribute of the class, named `attrName` and of type `attrType`, two cases are possible:

1) The attribute type is a single value; then it can be read with `attrType getAttrName()` and written with `void setAttrName(attrType a)`, where every call to `setAttrName()` overwrites any previous value of the attribute.
2) The attribute type is a set or a sequence of values; then there is an `void addAttrName(attrType a)` method to insert a new value and a `void clearAllAttrName()` method to remove all the values (the list becomes empty). Reading is performed by a `Iterator getAllAttrName()` method that returns an `Iterator` object that allows the programmer to walk through the `List` and cast its elements to the appropriate type.

Refer to the HTML documentation for a complete list of these classes and their interface.

3.1.1.1 Basic concepts of the ontology

The package `jade.onto.basic` includes a set of classes that are commonly part of every ontology, such as `Action`, `TruePredicate`, `FalsePredicate`, `ResultPredicate`, … The `BasicOntology` can be joined to any user-defined ontology as described in section 3.6.

Notice that the `Action` class should be used to represent actions. It has a couple of methods to set/get the AID of the actor (i.e. the agent who should perform the action) and the action itself (e.g. *Register*/*Deregister*/*Modify*).

3.1.2 Simplified API to access DF and AMS services

JADE features described so far allow complete interactions between FIPA system agents and user defined agents, simply by sending and receiving messages as defined by the standard.

However, because those interactions have been fully standardized and because they are very common, the following classes allow to successfully accomplish this task with a simplified interface.

Two methods are implemented by the class `Agent` to get the AID of the default DF and AMS of the platform: `getDefaultDF()` and `getAMS()`.

3.1.2.1 DFServiceCommunicator

`jade.domain.DFServiceCommunicator` implements a set of static methods to communicate with a standard FIPA DF service (i.e. a yellow pages agent).

It includes methods to request `register`, `deregister`, `modify` and `search` actions from a DF. Each of this method has a version with all the needed parameters, and one with a subset of them where the omitted parameters are given default values.

Notice that these methods block every agent activity until the action is successfully executed or a `jade.domain.FIPAException` exception is thrown (e.g. because a `failure` message has been received by the DF), that is, until the end of the conversation.

In some cases, instead, it is more convenient to execute this task in a non-blocking way. The method `getNonBlockingBehaviour()` returns a non-blocking behaviour (of type `RequestFIPAServiceBehaviour`) that can be added to the agent behaviours, as usual, by using `Agent.addBehaviour()`. Several ways are available to get the result of this behaviour and the programmer can select one according to his preferred programming style:

- call `getLastMsg()` and `getSearchResults()` (both methods throw a `NotYetReadyException` if the task has not yet finished);
- create a `SequentialBehaviour` composed of two sub-behaviours: the first one is the returned `RequestFIPAServiceBehaviour`, while the second one is application-dependent and is executed only when the first is terminated;
- use the class `RequestFIPAServiceBehaviour` by extending it and overriding all the `handleXXX()` methods that handle the states of the `fipa-request` interaction protocol.

3.1.2.2 AMSServiceCommunicator

This class is dual of `DFServiceCommunicator` class, accessing services provided by a standard FIPA AMS agent and its interface completely corresponds the the `DFServiceCommunicator` one.

Notice that JADE calls automatically the `register` and `deregister` methods with the default AMS respectively before calling `setup()` method and just after `takeDown()` method returns; so there is no need for a normal programmer to call them.

However, under certain circumstances, a programmer might need to call its methods. To give some examples: when an agent wishes to register with the AMS of a remote agent platform, or when an agent wishes to modify its description by adding a private address to the set of its addresses, …

3.2 The Agent class

The `Agent` class represents a common base class for user defined agents. Therefore, from the programmer's point of view, a JADE agent is simply an instance of a user defined Java class that extends the base `Agent` class. This implies the inheritance of features to accomplish basic interactions with the agent platform (registration, configuration, remote management, …) and a basic set of methods that can be called to implement the custom behaviour of the agent (e.g. send/receive messages, use standard interaction protocols, register with several domains, …).

The computational model of an agent is multitask, where tasks (or behaviours) are executed concurrently. Each functionality/service provided by an agent should be implemented as one or more behaviours (refer to section 3.4 for implementation of behaviours). A scheduler, internal to the base `Agent` class and hidden to the programmer, automatically manages the scheduling of behaviours.

3.2.1 Agent life cycle

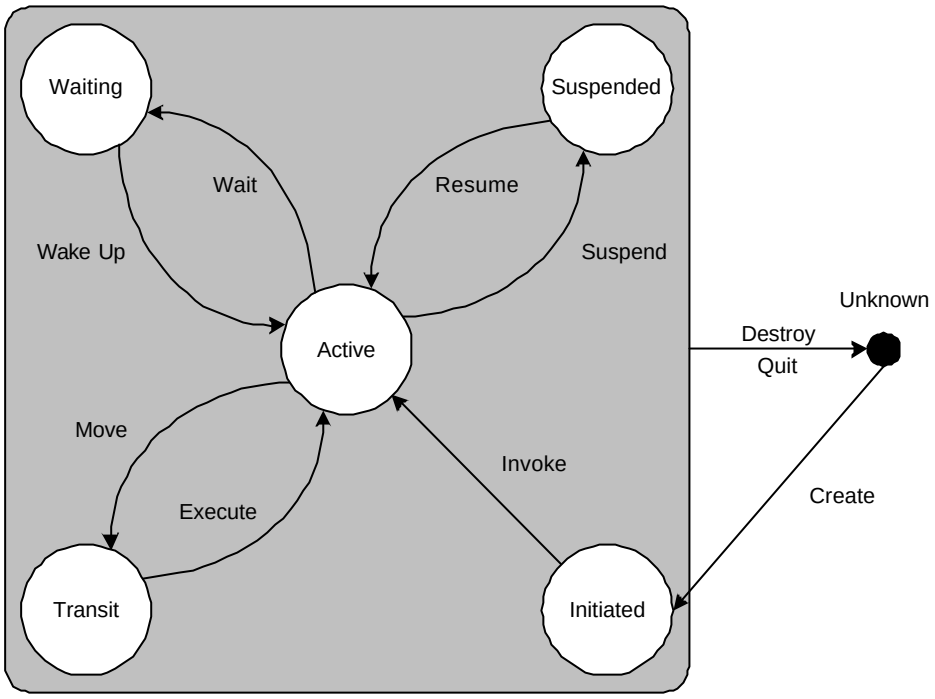

Figure 3 - Agent life-cycle as defined by FIPA.

A JADE agent can be in one of several states, according to Agent Platform Life Cycle in FIPA specification; these are represented by some constants in `Agent` class. The states are:

- **AP_INITIATED :** the Agent object is built, but hasn't registered itself yet with the AMS, has neither a name nor an address and cannot communicate with other agents.
- **AP_ACTIVE :** the Agent object is registered with the AMS, has a regular name and address and can access all the various JADE features.
- **AP_SUSPENDED :** the Agent object is currently stopped. Its internal thread is suspended and no agent behaviour is being executed.
- **AP_WAITING :** the Agent object is blocked, waiting for something. Its internal thread is sleeping on a Java monitor and will wake up when some condition is met (typically when a message arrives).
- **AP_DELETED :** the Agent is definitely dead. The internal thread has terminated its execution and the Agent is no more registered with the AMS.
- **AP_TRANSIT:** a mobile agent enters this state while it is migrating to the new location. The system continues to buffer messages that will then be sent to its new location.
- **AP_COPY:** this state is internally used by JADE for agent being cloned.
- **AP_GONE:** this state is internally used by JADE when a mobile agent has migrated to a new location and has a stable state.

The `Agent` class provides public methods to perform transitions between the various states; these methods take their names from a suitable transition in the Finite State Machine shown in FIPA specification *Agent Management*. For example, `doWait()` method puts the agent into AP_WAITING state from AP_ACTIVE state, `doSuspend()` method puts the agent into

AP_SUSPENDED state from AP_ACTIVE or AP_WAITING state, … Refer to the HTML documentation of the `Agent` class for a complete list of these `doXXX()` methods.

Notice that an agent is allowed to execute its behaviours (i.e. its tasks) only when it is in the AP_ACTIVE state. Take care that **if any behaviours call the `doWait()` method, then the whole agent and all its activities are blocked and not just the calling behaviour**. Instead, the `block()` method is part of the `Behaviour` class in order to allow suspending a single agent behaviour (see section 3.4 for details on the usage of behaviours).

3.2.1.1 Starting the agent execution

The JADE framework controls the birth of a new agent according to the following steps: the agent constructor is executed, the agent is given an identifier (see the HTML documentation for the `jade.core.AID` class), it is registered with the AMS, it is put in the AP_ACTIVE state, and finally the `setup()` method is executed. According to the FIPA specifications, an agent identifier has the following attributes:

- *a globally unique name*. By default JADE composes this name as the concatenation of the local name – i.e. the agent name provided on the command line – plus the '@' symbol, plus the home agent platform identifier – i.e. <hostname> ':' <port number of the JADE RMI registry> '/' 'JADE'); Tough in the case that the name of the platform is specified on the command line the agent name is constructed as a concatenation of the local name plus the '@' symbol plus the specified platform name.
- a set of agent addresses. Each agent inherits the transport addresses of its home agent platform;
- a set of resolvers, i.e. white page services with which the agent is registered.

The `setup()` method is therefore the point where any application-defined agent activity starts. The programmer has to implement the `setup()`method in order to initialise the agent. When the `setup()` method is executed, the agent has been already registered with the AMS and its Agent Platform state is *AP_ACTIVE*. The programmer should use this initialisation procedure to:

- (optional) if necessary, modify the data registered with the AMS (see section 3.1.2);
- (optional) set the description of the agent and its provided services and, if necessary, register the agent with one or more domains, i.e. DFs (see section 3.1.2);
- (necessary) add tasks to the queue of ready tasks using the method `addBehaviour()`. These behaviours are scheduled as soon as the `setup()` method ends;

The `setup()` method should add at least one behaviour to the agent. At the end of the `setup()` method, JADE automatically executes the first behaviour in the queue of ready tasks and then switch to the other behaviours in the queue by using a round-robin non-preemptive scheduler. The `addBehaviour(Behaviour)` and `removeBehaviour(Behaviour)` methods of the `Agent` class can be used to manage the task queue.

3.2.1.2 Stopping agent execution

Any behaviour can call the `Agent.doDelete()` method in order to stop agent execution.

The `Agent.takeDown()` method is executed when the agent is about to go to AP_DELETED state, i.e. it is going to be destroyed. The `takeDown()` method can be overridden by the programmers in order to implement any necessary cleanup. When this method is executed the agent is still registered with the AMS and can therefore send messages to other

agents, but just after the `takeDown()` method is completed, the agent will be de-registered and its thread destroyed. The intended purpose of this method is to perform application specific cleanup operations, such as de-registering with DF agents.

3.2.2 Inter-agent communication.

The `Agent` class also provides a set of methods for inter-agent communication. According to the FIPA specification, agents communicate via asynchronous message passing, where objects of the `ACLMessage` class are the exchanged payloads. See also section 3.3 for a description of the `ACLMessage` class. Some of the interaction protocols defined by FIPA are also available as ready-to-use behaviours that can be scheduled for agent activities; they are part of the `jade.proto` package.

The `Agent.send()` method allows to send an `ACLMessage`. The value of the `receiver` slot holds the list of the receiving agent IDs. The method call is completely transparent to where the agent resides, i.e. be it local or remote, it is the platform that takes care of selecting the most appropriate address and transport mechanism.

3.2.2.1 Accessing the private queue of messages.

All the messages received by an agent are put in its private queue by the agent platform. Several access modes have been implemented in order to get messages from this private queue:

- The message queue can be accessed in a blocking (using `blockingReceive()` method) or non-blocking way (using `receive()` method). The blocking version must be used very carefully because it ***causes the suspension of all the agent activities and in particular of all its Behaviours.*** The non-blocking version returns immediately `null` when the requested message is not present in the queue;
- both methods can be augmented with a pattern-matching capability where a parameter is passed that describes the pattern of the requested `ACLMessage`. Section 3.3.4 describes the `MessageTemplate` class;
- the blocking access can have a timeout parameter. It is a *long* that describes the maximum number of milliseconds that the agent activity should remain blocked waiting for the requested message. If the timeout elapses before the message arrives, the method returns `null`;
- the two behaviours `ReceiverBehaviour` and `SenderBehaviour` can be used to schedule agent tasks that requires receiving or sending messages.

3.2.3 Agents with a graphical user interface (GUI).

An application, that is structured as a Multi Agent System, still needs to interact with its users. So, it is often necessary to provide a GUI for at least some agents in the application. This need raises some problems, though, stemming from the mismatch between the autonomous nature of agents and the reactive nature of ordinary graphical user interfaces. When JADE is used, the *thread-per-agent* concurrency model of JADE agents must work together with the Swing concurrency model.

3.2.3.1 Java GUI concurrency model

In a Java Virtual Machine there is a single thread, called *Event Dispatcher Thread*, whose task is to continuously pick event objects (i.e. instances of `java.awt.AWTEvent` class) from the *System Event Queue* (which is an instance of `java.awt.EventQueue` class). Then the event dispatcher thread, among other things, calls the various listeners registered with the event source. The important observation is that all event listeners are executed within a single thread of control (the event dispatcher); from this follows the well known rule that the execution time of an event listener should be short (less than 0.1 s) to ensure interface responsiveness.
A very important Swing feature is the *Model/View* system to manage GUI updates. When a Swing control has some state (a `JCheckBox` has a `checked` flag, a `JList` holds elements, etc.), this state is kept in a *Model* object (of class `DefaultButtonModel`, `ListModel`, etc.). The model object provides commands to modify the state (e.g. to check or uncheck the checkbox, to add and remove elements from the list, etc.) and the Swing built-in notification mechanism updates the visual appearance of the GUI to reflect the state change. So, a `JCheckBox` object can change its look in two cases:

- An event from the user is received (e.g. a `MouseClick` event).
- Some other part of the program modifies the model object associated with the `JCheckBox`.

As stated in the *Java Tutorial* (*JFC/Swing* trail, *Threads and Swing* section), the Swing framework is not thread-safe, so any code that updates the GUI elements must be executed within the event dispatcher thread; since modifying a model object triggers an update of the GUI, it follows from the above that model objects also have to be manipulated just by the event dispatcher thread. The Swing framework provides a simple but general way to pass some user defined code to the Event Dispatcher thread: the `SwingUtilities` class exposes two static methods that accept a `Runnable` object, wrap it with a `RunnableEvent` and push it into the System Event Queue. The `invokeLater()` method puts the `Runnable` into the System Event Queue and returns immediately (behaving like an asynchronous inter-thread call), whereas the `invokeAndWait()` method puts the `Runnable` into the System Event Queue and blocks until the Event Dispatcher thread has processed the `RunnableEvent` (behaving like a synchronous inter-thread call). Moreover, the `invokeAndWait()` method can catch exceptions thrown within the `Runnable` object.

3.2.3.2 Performing an ACL message exchange in response to a GUI event.

When an agent is given a GUI, it often happens that the agent is requested to send a message because of a user action (e.g., the user clicks a pushbutton). The `ActionListener` of the button will be run within the Event Dispatcher thread, but the `Agent.send()` method should be called within the agent thread.

So:

In the event listener, add a new behaviour to the agent, which performs the necessary communication.

If the communication to perform is simply a message send operation, the `SenderBehaviour` class can be used, and the event handler will contain a line such as:

```
myAgent.addBehaviour(new SenderBehaviour(msgToSend));
```

If the communication operation is a message receive, the `ReceiverBehaviour` class can be used in the same way:

```
myAgent.addBehaviour(new ReceiverBehaviour(msgToRecv));
```

More generally, some complex conversation (e.g. a whole interaction conforming to an Interaction Protocol) could be started when the user acts on the GUI. The solution, again, is to add a new behaviour to the agent; this behaviour will extend the predefined JADE behaviours for Interaction Protocols or will be a custom complex behaviour.
The following code is extracted from the JADE RMA management agent. When the user wants to create a new agent, he or she operates on the RMA GUI (through the menu bar, the tool bar or a popup menu) to cause the execution of a `StartNewAgentAction` object, which calls the `newAgent()` method of the `rma` class. This method is implemented as follows:

```
      public void newAgent(String agentName, String className, Object
arg[], String containerName) {
        // Create a suitable content object for the ACL message ...
        // Set the :ontology slot of the message
    requestMsg.setOntology(JADEAgentManagementOntology.NAME);
        // Fill the message content with a List l, containing the
content object
    fillContent(requestMsg, l);
        addBehaviour(new AMSClientBehaviour("CreateAgent", requestMsg));
      }
```

The `AMSClientBehaviour` class is a private inner class of the `rma` class, that extends the `FipaRequestInitiatorBehaviour` and plays the *fipa-request* Interaction Protocol with the AMS agent. In this case, the `addBehaviour()` call and the specific class of the behaviour to add are completely encapsulated into the `rma` class. Various classes of the RMA GUI (mainly the action classes) hold a reference to the RMA agent and use it to call methods such as `newAgent()`. Notice that methods such as `newAgent()`don't really belong to the agent, because they don't access the agent state in any way. So, they are designed for being called from the outside (a different execution thread): in the following, these methods will be called *external methods*.
In general, it is not a good thing that an external software component maintain a direct object reference to an agent, because this component could directly call any public method of the agent (not just the external ones), skipping the asynchronous message passing layer and turning an autonomous agent into a server object, slave to its caller.
A better approach would be to gather all the external methods into an interface, implemented by the agent class. Then, an object reference of that interface will be passed to the GUI so that only the external methods will be callable from event handlers. The following pseudo code illustrates this approach:

```
interface RMAExternal {
  void newAgent(String agentName, String className, Object arg[], String
containerName);
  void suspendAgent(AID name);
  void resumeAgent(AID name);
  void killAgent(AID name);
  void killContainer(String name);
  void shutDownPlatform();
}
class MainWindow extends JFrame {
  private RMAExternal myRMA;
      public MainWindow(RMAExternal anRMA) {
    myRMA = anRMA;
  }
```

```
      // ...
}
class rma extends Agent implements RMAExternal {
  private MainWindow myGUI;
  protected void setup() {
    myGUI = new MainWindow(this);//Parameter 'this' typed as RMAExternal
    // ...
  }
}
```

With the schema above, the GUI will be able to call only the external methods of the RMA agent.

3.2.3.3 Modifying the GUI when an ACL message is received.

An agent can receive information from other agents through ACL messages: the *inform* FIPA communicative act serves just this purpose. If the agent has a GUI, it may often be the case that it wants to communicate the new information to its user by modifying the visual appearance of the GUI. According to the Model/View pattern, the new information should be used to modify some model objects, and Swing will take automatically care of updating the GUI. The `Agent.receive()` operation that read the message was executed within the agent thread, but any modification to Swing model objects must be performed from the Event Dispatcher thread. So:

In the agent behaviour, encapsulate all access to GUI model objects into a `Runnable` object and use `SwingUtilities.invokeXXX()` to submit the `Runnable` to the Event Dispatcher thread.

For example, when a new agent is born on a JADE platform, the AMS sends *inform* messages to all the active RMA agents; each one of them has to update its `AgentTree`, adding a node representing the new agent. The `rma` class holds a behaviour of the (inner and private) `AMSListener` class that continously receives `inform` messages from the AMS and dispatches them to suitable internal event handlers (it is basically a simple distributed event system over ACL messages). The handler corresponding to the *agent-born* event has the following code:

```
public void handle(AMSEvent ev) {
  AgentBorn ab = (AgentBorn)ev;
  String container = ab.getContainer();
  AID agent = ab.getAgent();
  myGUI.addAgent(container, agent);
}
```

The `addAgent()` method of the class `MainWindow` is the following:

```
public void addAgent(final String containerName, final AID agentID) {
  // Add an agent to the specified container
  Runnable addIt = new Runnable() {
  public void run() {
    String agentName = agentID.getName();
    AgentTree.Node node = tree.treeAgent.createNewNode(agentName, 1);
    Iterator add = agentID.getAllAddresses();
    String agentAddresses = "";
    while(add.hasNext())
      agentAddresses = agentAddresses + add.next() + " ";
      tree.treeAgent.addAgentNode((AgentTree.AgentNode)node,
containerName, agentName, agentAddresses, "FIPAAGENT");
```

```
    }
  };
  SwingUtilities.invokeLater(addIt);
}
```

As can be seen from the above code, all the accesses to the agent tree are encapsulated inside a `Runnable` that is submitted for execution to the Event Dispatcher thread using the `SwingUtilities.invokeLater()` method. The whole process of `Runnable` creation and submission is contained within the `addAgent()` method of the `MainWindow` class, so that the rma agent does not directly deal with Swing calls (it does not even have to import Swing related classes).
If we consider the whole `MainWindow` as an active object whose thread is the Event Dispatcher thread, then the `addAgent()` method is clearly an external method and this approach mirrors exactly the technique used in the section above. However, since the GUI is not to be seen as an autonomous software component, the choice of using external methods or not is just a matter of software structure, without particular conceptual meaning.

3.2.3.4 Support for building GUI enabled agents in JADE.

Because it is quite common having agents with a GUI, JADE includes the class `jade.gui.GuiAgent` for this specific purpose. This class is a simple extension of the `jade.core.Agent` class: at the start-up (i.e. when the method `setup()` is executed) it instantiates an ad-hoc behaviour that manages a queue of `jade.gui.GuiEvent` event objects that can be received by other threads. This behaviour is of course hidden to the programmer who needs only to implement the application-specific code relative to each event. In detail, the following operations must be performed.

A thread (in particular the GUI) wishing to notify an event to an agent should create a new object of type `jade.gui.GuiEvent` and pass it as a parameter to the call of the method `postGuiEvent()` of the `jade.gui.GuiAgent` object. After the method `postGuiEvent()` is called, the agent reacts by waking up all its active behaviours, and in particular the behaviour above mentioned that causes the agent thread to execute the method `onGuiEvent()`. Notice that an object `GuiEvent` has two mandatory attributes (i.e. the source of the event and an integer identifying the type of event) and an optional list of parameters[4] that can be added to the event object.

As a consequence, an agent wishing to receive events from another thread (in particular its GUI) should define the types of events it intends to receive and then implement the method `onGuiEvent()`. In general, this method is a big switch, one case for each type of event. The example *mobile*, distributed with JADE, is a good example of this feature.

In order to explain further the previous concepts, in the following are reported some interesting points of the code of the example concerning the `MobileAgent`.

File MobileAgent.java

[4] The type of each parameter must extend *java.lang.Object;* therefore primitive objects (e.g. *int*) should before be wrapped into appropriate objects (e.g. *java.lang.Integer).*

```
public class MobileAgent extends GuiAgent {

  ……

  // These constants are used by the Gui to post Events to the
  Agent
  public static final int EXIT = 1000;
  public static final int MOVE_EVENT = 1001;
  public static final int STOP_EVENT = 1002;
  public static final int CONTINUE_EVENT = 1003;
  public static final int REFRESH_EVENT = 1004;
  public static final int CLONE_EVENT = 1005;

  ……
public void setup() {
    ……
    // creates and shows the GUI
    gui = new MobileAgentGui(this);
    gui.setVisible(true);
    ……
  }
  ……

  // AGENT OPERATIONS FOLLOWING GUI EVENTS
  protected void onGuiEvent(GuiEvent ev)
  {
       switch(ev.getType())
       {
       case EXIT:
            gui.dispose();
            gui = null;
            doDelete();
            break;
       case MOVE_EVENT:
             Iterator moveParameters = ev.getAllParameter();
             nextSite =(Location)moveParameters.next();
            doMove(nextSite);
            break;
       case CLONE_EVENT:
            Iterator cloneParameters = ev.getAllParameter();
            nextSite =(Location)cloneParameters.next();
```

```
                doClone(nextSite,"clone"+cnt+"of"+getName());
                break;
          case STOP_EVENT:
                stopCounter();
                break;
          case CONTINUE_EVENT:
                continueCounter();
                break;
          case REFRESH_EVENT:
                addBehaviour(new
                GetAvailableLocationsBehaviour(this));
                break;
          }

    }
}
```

File MobileAgentGui.java

```
package examples.mobile;

public class MobileAgentGui extends JFrame implements
ActionListener
{
  private MobileAgent  myAgent;
  ……

  // Constructor
  MobileAgentGui(MobileAgent a)
  {
        super();
        myAgent = a;

        ……

        JButton pauseButton = new JButton("STOP COUNTER");
        pauseButton.addActionListener(this);
        Jbutton continueButton = new JButton("CONTINUE
        COUNTER");
        continueButton.addActionListener(this);
        …
        JButton b = new JButton(REFRESHLABEL);
```

```
        b.addActionListener(this);
        …
        b = new JButton(MOVELABEL);
        b.addActionListener(this);
        …
        b = new JButton(CLONELABEL);
        b.addActionListener(this);
        …
        b = new JButton(EXITLABEL);
        b.addActionListener(this);
        ……
    }
      ……

    public void actionPerformed(ActionEvent e)
    {
        String command = e.getActionCommand();
        // MOVE
        if (command.equalsIgnoreCase(MOVELABEL)) {
          Location dest;
          int sel = availableSiteList.getSelectedRow();
          if (sel >= 0)
            dest = availableSiteListModel.getElementAt(sel);
          else
            dest = availableSiteListModel.getElementAt(0);

        GuiEvent ev = new
        GuiEvent((Object)this,myAgent.MOVE_EVENT);
        ev.addParameter(dest);
         myAgent.postGuiEvent(ev);
        }
        // CLONE
        else if (command.equalsIgnoreCase(CLONELABEL)) {
          Location dest;
          int sel = availableSiteList.getSelectedRow();
          if (sel >= 0)
            dest = availableSiteListModel.getElementAt(sel);
          else
            dest = availableSiteListModel.getElementAt(0);
          GuiEvent ev = new
                GuiEvent((Object)this,myAgent.CLONE_EVENT);
```

```
            ev.addParameter(dest);
                 myAgent.postGuiEvent(ev);
          }
          // EXIT
          else if (command.equalsIgnoreCase(EXITLABEL)) {
                GuiEvent ev = new GuiEvent(null,myAgent.EXIT);
                myAgent.postGuiEvent(ev);
          }
          else if (command.equalsIgnoreCase(PAUSELABEL)) {
             GuiEvent ev = new GuiEvent(null,myAgent.STOP_EVENT);
             myAgent.postGuiEvent(ev);
          }
          else if (command.equalsIgnoreCase(CONTINUELABEL)){
       GuiEvent ev = new GuiEvent(null,myAgent.CONTINUE_EVENT);
        myAgent.postGuiEvent(ev);
       }
          else if (command.equalsIgnoreCase(REFRESHLABEL)) {
      GuiEvent ev = new GuiEvent(null,myAgent.REFRESH_EVENT);
      myAgent.postGuiEvent(ev);
          }
      }
      ……
  }
```

3.2.4 Agent with parameters and launching agents

A list of arguments can be passed to an `Agent` and they can be retrieved by calling the method `Object[] getArguments()`. Notice that the arguments are transient and they do not migrate with the agent neither they are cloned with the agent.

There are three ways of launching an agent:

- a list of agents can be specified on the command line, by using the syntax described in the Administrator's Guide. Arguments, embedded within parenthesis, can be passed to each agent. This is the most common option and the option that best matches the theoretical requirement of agent autonomy.

- an agent can be launched by an administrator by using the RMA (Remote Monitoring Agent) GUI, as described in the Administrator's Guide. Arguments, embedded within parenthesis, can be passed to each agent.

- finally, an agent can also be launched by any external Java program by using the InProcess Interface as described in section 3.8

3.3 Agent Communication Language (ACL) Messages

The class `ACLMessage` represents ACL messages that can be exchanged between agents. It contains a set of attributes as defined by the FIPA specifications.

An agent willing to send a message should create a new `ACLMessage` object, fill its attributes with appropriate values, and finally call the method `Agent.send()`. Likewise, an agent willing to receive a message should call `receive()` or `blockingReceive()` methods, both implemented by the `Agent` class and described in section 3.2.2.

Sending or receiving messages can also be scheduled as independent agent activities by adding the behaviours `ReceiverBehaviour` and `SenderBehaviour` to the agent queue of tasks.

All the attributes of the `ACLMessage` object can be accessed via the `set/get<Attribute>()` access methods. All attributes are named after the names of the parameters, as defined by the FIPA specifications. Those parameters whose type is a set of values (like receiver, for instance) can be accessed via the methods `add/getAll<Attribute>()` where the first method adds a value to the set, while the second method returns an `Iterator` over all the values in the set. Notice that all the `get()` methods return `null` when the attribute has not been yet set.

Furthermore, this class also defines a set of constants that should be used to refer to the FIPA performatives, i.e. `REQUEST`, `INFORM`, etc. When creating a new `ACLMessage` object, one of these constants must be passed to `ACLMessage` class constructor, in order to select the message performative. The `reset()` method resets the values of all message fields.

The `toString()` method returns a string representing the message. This method should be just used for debugging purposes.

3.3.1 Support to reply to a message

According to FIPA specifications, a reply message must be formed taking into account a set of well-formed rules, such as setting the appropriate value for the attribute *in-reply-to*, using the same *conversation-id*, etc. JADE helps the programmer in this task via the method `createReply()` of the `ACLMessage` class. This method returns a new `ACLMessage` object that is a valid reply to the current one. Then, the programmer only needs to set the application-specific communicative act and message content.

3.3.2 Support for Java serialisation and transmission of a sequence of bytes

Some applications may benefit from transmitting a sequence of bytes over the content of an `ACLMessage`. A typical usage is passing Java objects between two agents by exploiting the Java serialization. The `ACLMessage` class supports the programmer in this task by allowing the usage of *Base64* encoding through the two methods `setContentObject()` and `getContentObject()`. Refer to the HTML documentation of the JADE API and to the examples in `examples/Base64` directory for an example of usage of this feature.

It must be noticed that this feature does not comply to FIPA and that any agent platform can recognize automatically the usage of Base64 encoding[5], so the methods must appropriately used

[5] The implementation of this feature uses the source code contained within the src/starlight directory. This code is covered by the GNU General Public License, as decided by the copyright owner Kevin Kelley. The GPL license itself has been included as a text file named COPYING in the same directory. If the programmer does not need any support for Base64 encoding, then this code is not necessary and can be removed.

by the programmers and should suppose that communicating agents know a-priori the usage of these methods.

3.3.3 The ACL Codec

Under normal conditions, agents never need to call explicitly the codec of the ACL messages because it is done automatically by the platform. However, when needed for some special circumstances, the programmer should use the methods provided by the class `StringACLCodec` to parse and encode ACL messages in String format.

3.3.4 The MessageTemplate class

The JADE behaviour model allows an agent to execute several parallel tasks. However any agent should be provided with the capability of carrying on also many simultaneous conversations. Because the queue of incoming messages is shared by all the agent behaviours, an access mode to that queue based on pattern matching has been implemented (see 3.2.2.1).

The `MessageTemplate` class allows to build patterns to match ACL messages against. Using the methods of this class the programmer can create one pattern for each attribute of the `ACLMessage`. Elementary patterns can be combined with AND, OR and NOT operators, in order to build more complex matching rules. In such a way, the queue of incoming ACL messages can be accessed via pattern-matching rather than FIFO.

The user can also define application specific patterns extending the MatchExpression interface in order to provide a new `match()` method to use in the pattern matching phase.

The example WaitAgent in the MessageTemplate directory of the package examples, shows a way to create an application-specific MessageTemplate:

```
public class WaitAgent extends Agent {
  Class myMatchExpression implements
  MessageTemplate.MatchExpression {
    List senders;
    myMatchExpression(List l){
       senders = l;
    }
    public boolean match(ACLMessage msg){
      AID sender = msg.getSender();
      String name = sender.getName();
      Iterator it_temp = senders.iterator();
      boolean out = false;

      while(it_temp.hasNext() && !out){
       String tmp = ((AID)it_temp.next()).getName();
       if(tmp.equalsIgnoreCase(name))
             out = true;
      }
      return out;
```

```
            }

      }
      class WaitBehaviour extends SimpleBehaviour {
        public WaitBehaviour(Agent a, MessageTemplate mt) {
            ……}
        public void action() {
            ……
            ACLMessage  msg = blockingReceive(template);
            ……
        }
        ……
      } //End class WaitBehaviour

      protected void setup() {
             ……
            ArrayList sender = ……
            myMatchExpression me = new myMatchExpression(sender);
            MessageTemplate myTemplate = new MessageTemplate(me);

            MessageTemplate mt =
MessageTemplate.and(myTemplate,MessageTemplate.MatchPerformative(
ACLMessage.REQUEST));

            WaitBehaviour behaviour = new  WaitBehaviour(this,mt);
            addBehaviour(behaviour);
            }catch(java.io.IOException e){
                      e.printStackTrace();
      }
      }
   }//end class WaitAgent
```

3.4 The agent tasks. Implementing Agent behaviours

An agent must be able to carry out several concurrent tasks in response to different external events. In order to make agent management efficient, every JADE agent is composed of a single execution thread and all its tasks are modelled and can be implemented as `Behaviour` objects. Multi-threaded agents can also be implemented but no specific support (except synchronizing the ACL message queue) is provided by JADE.

The developer who wants to implement an agent-specific task should define one or more `Behaviour` subclasses, instantiate them and add the behaviour objects to the agent task list. The `Agent` class, which must be extended by agent programmers, exposes two methods: `addBehaviour(Behaviour)` and `removeBehaviour(Behaviour)`, which allow to

manage the ready tasks queue of a specific agent. Notice that behaviours and sub-behaviours can be added whenever is needed, and not only within `Agent.setup()` method. Adding a behaviour should be seen as a way to spawn a new (cooperative) execution thread within the agent.

A scheduler, implemented by the base `Agent` class and hidden to the programmer, carries out a round-robin non-preemptive scheduling policy among all behaviours available in the ready queue, executing a `Behaviour`-derived class until it will release control (this happens when `action()` method returns). If the task relinquishing the control has not yet completed, it will be rescheduled the next round. A behaviour can also block, waiting for a message to arrive. In detail, the agent scheduler executes `action()` method of each behaviour present in the ready behaviours queue; when `action()` returns, the method `done()` is called to check if the behaviour has completed its task. If so, the behaviour object is removed from the queue.

Behaviours work just like co-operative threads, but there is no stack to be saved. ***Therefore, the whole computation state must be maintained in instance variables of the Behaviour and its associated Agent***.

In order to avoid an active wait for messages (and, as a consequence, a waste of CPU time), every single `Behaviour` is allowed to block its computation. ***The method `block()` puts the behaviour in a queue of blocked behaviours as soon as the `action()` method returns.*** Notice, therefore, that the blocking effect is not achieved immediately after calling the `block()` method, but just after returning from the `action()` method. All blocked behaviours are rescheduled as soon as a new message arrives, therefore the programmer must take care of blocking again a behaviour if it was not interested in the arrived message. Moreover, a behaviour object can block itself for a limited amount of time passing a timeout value to `block()` method. In future releases of JADE, more wake up events will be probably considered.

Because of the non preemptive multitasking model chosen for agent behaviours, agent programmers must avoid to use endless loops and even to perform long operations within `action()` methods. Remember that when some behaviour's `action()` is running, no other behaviour can go on until the end of the method (of course this is true only with respect to behaviours of the same agent: behaviours of other agents run in different Java threads and can still proceed independently).

Besides, since no stack contest is saved, every time `action()` method is run from the beginning: there is no way to interrupt a behaviour in the middle of its `action()`, yield the CPU to other behaviours and then start the original behaviour back from where it left.

For example, suppose a particular operation `op()` is too long to be run in a single step and is therefore broken in three sub-operations, named `op1()`,`op2()` and `op3()`. To achieve desired functionality one must call `op1()` the first time the behaviour is run, `op2()` the second time and `op3()` the third time, after which the behaviour must be marked as terminated. The code will look like the following:

```
public class my3StepBehaviour {
  private int state = 1;
  private boolean finished = false;

  public void action() {
    switch (state) {
      case 1: { op1(); state++; break; }
```

```
        case 2: { op2(); state++; break; }
        case 3: { op3(); state=1; finished = true; break; }
       }
    }

    public boolean done() {
      return finished;
    }
  }
```

Following this idiom, agent behaviours can be described as finite state machines, keeping their whole state in their instance variables.

When dealing with complex agent behaviours (as agent interaction protocols) using explicit state variables can be cumbersome; so JADE also supports a compositional technique to build more complex behaviours out of simpler ones.

The framework provides ready to use `Behaviour` subclasses that can contain sub-behaviours and execute them according to some policy. For example, a `SequentialBehaviour` class is provided, that executes its sub-behaviours one after the other for each `action()` invocation.

The following figure is an annotated UML class diagram for JADE behaviours.

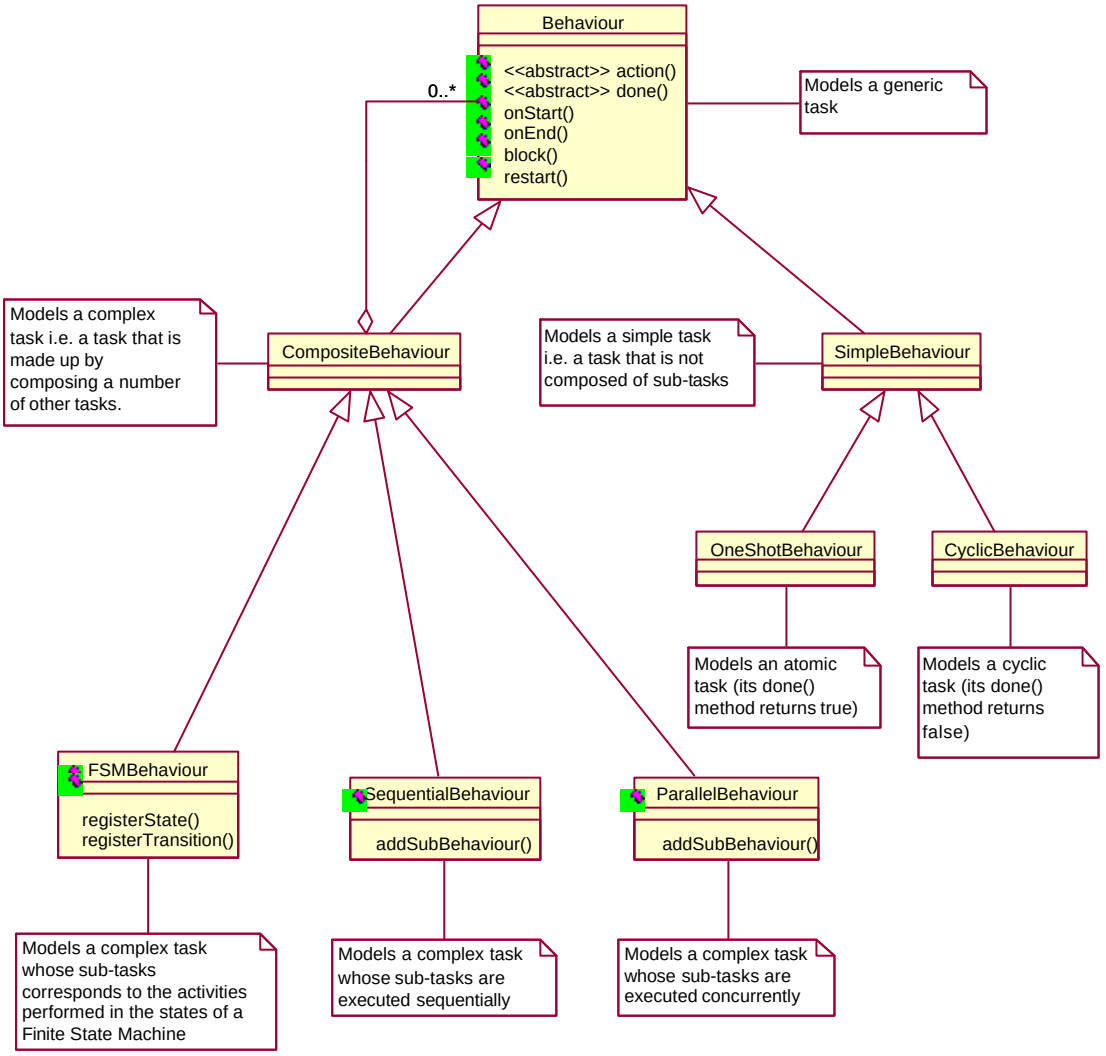

Figure 4 - UML Model of the Behaviour class hierarchy

Starting from the basic class `Behaviour`, a class hierarchy is defined in the `jade.core.behaviour` package of the JADE framework.
A complete description of all these classes follows.

3.4.1 class Behaviour

This abstract class provides an abstract base class for modelling agent tasks, and it sets the basis for behaviour scheduling as it allows for state transitions (i.e. starting, blocking and restarting a Java behaviour object).

The `block()` method allows to block a behaviour object until some event happens (typically, until a message arrives). This method leaves unaffected the other behaviours of an agent, thereby allowing finer grained control on agent multitasking. This method puts the behaviour in a queue of blocked behaviours and takes effect as soon as `action()` returns. All blocked behaviours are rescheduled as soon as a new message arrives. Moreover, a behaviour

object can block itself for a limited amount of time passing a timeout value to `block()` method, expressed in milliseconds. In future releases of JADE, more wake up events will be probably considered. A behaviour can be explicitly restarted by calling its `restart()` method.

Summarizing, a blocked behaviour can resume execution when one of the following three conditions occurs:

1. An ACL message is received by the agent this behaviour belongs to.
2. A timeout associated with this behaviour by a previous `block()` call expires.
3. The `restart()` method is explicitly called on this behaviour.

The Behaviour class also provides two placeholders methods, named `onStart()` and `onEnd()`. These methods can be overridden by user defined subclasses when some actions are to be executed before and after running behaviour execution.

onEnd() returns an int that represents a termination value for the behaviour.

It should be noted that onEnd() is called after the behaviour has completed and has been removed from the pool of agent behaviours. Therefore calling reset() inside onEnd() is not sufficient to cyclically repeat the task represented by that behaviour; besides that the behaviour should be added again to the agent as in the following example

```
public int onEnd() {
  reset();
  myAgent.addBehaviour(this);
  return 0;
}
```

This class provides also a couple of methods to get and set a *DataStore* for the behaviour. The *DataStore* can be a useful repository for exchanging data between behaviours, as done, for instance, by the classes *jade.proto.AchieveREInitiator/Responder*. **Notice that the DataStore is cleaned and all the contained data are lost when the behaviour is reset.**

3.4.2 class SimpleBehaviour

This abstract class models simple atomic behaviours. Its `reset()` method does nothing by default, but it can be overridden by user defined subclasses.

3.4.3 class OneShotBehaviour

This abstract class models atomic behaviours that must be executed only once and cannot be blocked. So, its `done()` method always returns `true`.

3.4.4 class CyclicBehaviour

This abstract class models atomic behaviours that must be executed forever. So its `done()` method always returns `false`.

3.4.5 class CompositeBehaviour

This abstract class models behaviours that are made up by composing a number of other behaviours (children). So the actual operations performed by executing this behaviour are not defined in the behaviour itself, but inside its children while the composite behaviour takes only

care of children scheduling according to a given policy[6].

In particular the CompositeBehaviour class only provides a common interface for children scheduling, but does not define any scheduling policy. This scheduling policy must be defined by subclasses (SequentialBehaviour, ParallelBehaviour and FSMBehaviour). A good programming practice is therefore to use only CompositeBehaviour sub-classes, unless some special children scheduling policy is needed (e.g. a `PriorityBasedCompositeBehaviour` should extend `CompositeBehaviour` directly).

Notice that this class was renamed since JADE 2.2 and it was previously called `ComplexBehaviour.`

3.4.6 class SequentialBehaviour

This class is a `CompositeBehaviour` that executes its sub-behaviours sequentially and terminates when all sub-behaviours are done. Use this class when a complex task can be expressed as a sequence of atomic steps (e.g. do some computation, then receive a message, then do some other computation).

3.4.7 class ParallelBehaviour

This class is a `CompositeBehaviour` that executes its sub-behaviours concurrently and terminates when a particular condition on its sub-behaviours is met. Proper constants to be indicated in the constructor of this class are provided to create a `ParallelBehaviour` that ends when all its sub-behaviours are done, when any one among its sub-behaviour terminates or when a user defined number *N* of its sub-behaviours have finished. Use this class when a complex task can be expressed as a collection of parallel alternative operations, with some kind of termination condition on the spawned subtasks.

Notice that this class was renamed since JADE 2.2 and it was previously called `NonDeterministicBehaviour.`

3.4.8 class FSMBehaviour

This class is a `CompositeBehaviour` that executes its children according to a Finite State Machine defined by the user. More in details each child represents the activity to be performed within a state of the FSM and the user can define the transitions between the states of the FSM. When the child corresponding to state S_i completes, its termination value (as returned by the onEnd() method) is used to select the transition to fire and a new state S_j is reached. At next round the child corresponding to S_j will be executed. Some of the children of an FSMBehaviour can be registered as final states. The FSMBehaviour terminates after the completion of one of these children.

Refer to the javadoc documentation of the JADE APIs for a detailed description on how to describe a Finite State Machine both at execution-time or static compilation time.

[6] Each time the `action`() method of a complex behaviour is called this results in calling the `action`() method of one of its children. The scheduling policy determines which children to select at each round.

3.4.9 class SenderBehaviour

Encapsulates an atomic unit which realises the "send" action. It extends `OneShotBehaviour` class and so it is executed only once. An object with this class must be given the ACL message to send at construction time.

3.4.10 class ReceiverBehaviour

Encapsulates an atomic operation which realises the "receive" action. Its action terminates when a message is received. If the message queue is empty or there is no message matching the `MessageTemplate` parameter, `action()` method calls `block()` and returns. The received message is copied into a user specified `ACLMessage`, passed in the constructor. Two more constructors take a timeout value as argument, expressed in milliseconds; a `ReceiverBehaviour` created using one of these two constructors will terminate after the timeout has expired, whether a suitable message has been received or not. An `Handle` object is used to access the received ACL message; when trying to retrieve the message suitable exceptions can be thrown if no message is available or the timeout expired without any useful reception.

3.4.11 class WakerBehaviour

This abstract class implements a OneShot task that must be executed only once just after a given timeout is elapsed.

3.4.12 Examples

In order to explain further the previous concepts, an example is reported in the following. It illustrates the implementation of two agents that, respectively, send and receive messages. The behaviour of the `AgentSender` extend the `SimpleBehaviour` class so it simply sends some messages to the receiver and than kills itself. The `AgentReceiver` has instead a behaviour that extends `CyclicBehaviour` class and shows different kinds to receive messages.

File AgentSender.java

```
package examples.receivers;

import java.io.*;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;

public class AgentSender extends Agent {

  protected void setup() {
    addBehaviour(new SimpleBehaviour(this) {
      private boolean finished = false;
      public void action() {
        try{
```

```
            System.out.println("\nEnter responder agent name: ");
            BufferedReader buff = new BufferedReader(new
                    InputStreamReader(System.in));
            String responder = buff.readLine();
            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            msg.addReceiver(new AID(responder));
            msg.setContent("FirstInform");
            send(msg);
            System.out.println("\nFirst INFORM sent");
            doWait(5000);
            msg.setLanguage("PlainText");
            msg.setContent("SecondInform");
            send(msg);
            System.out.println("\nSecond INFORM sent");
            doWait(5000);
           // same that second
           msg.setContent("\nThirdInform");
           send(msg);
           System.out.println("\nThird INFORM sent");
           doWait(1000);
           msg.setOntology("ReceiveTest");
           msg.setContent("FourthInform");
           send(msg);
           System.out.println("\nFourth INFORM sent");
           finished = true;
           myAgent.doDelete();
          }catch (IOException ioe){
           ioe.printStackTrace();
          }
        }
        public boolean done(){
           return finished;
        }
    });
    }
}
```

File AgentReceiver.java

```
package examples.receivers;

import java.io.*;
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
```

```
    public class AgentReceiver extends Agent {
      class my3StepBehaviour extends SimpleBehaviour {
       final int FIRST = 1;
       final int SECOND = 2;
       final int THIRD = 3;
       private int state = FIRST;
       private boolean finished = false;
        public my3StepBehaviour(Agent a) {
          super(a);
        }
        public void action() {
          switch (state){
           case FIRST: {if (op1())
                         state = SECOND;
                       else
                         state= FIRST;
                       break;}
           case SECOND:{op2(); state = THIRD; break;}
           case THIRD:{op3(); state = FIRST; finished = true; break;}
          }
        }

        public boolean done() {
          return finished;
        }


        private boolean op1(){
         System.out.println( "\nAgent "+getLocalName()+" in state 1.1 is
waiting for a message");
         MessageTemplate m1 =
            MessageTemplate.MatchPerformative(ACLMessage.INFORM);
         MessageTemplate m2 =
            MessageTemplate.MatchLanguage("PlainText");
         MessageTemplate m3 =
            MessageTemplate.MatchOntology("ReceiveTest");
         MessageTemplate m1andm2 = MessageTemplate.and(m1,m2);
         MessageTemplate notm3 = MessageTemplate.not(m3);
         MessageTemplate m1andm2_and_notm3 =
            MessageTemplate.and(m1andm2, notm3);

        //The agent waits for a specific message. If it doesn't arrive
        // the behaviour is suspended until a new message arrives.
        ACLMessage msg = receive(m1andm2_and_notm3);
```

```
        if (msg!= null){
          System.out.println("\nAgent "+ getLocalName() +
             " received the following message in state 1.1: " +
             msg.toString());
          return true;
        }
        else {
          System.out.println("\nNo message received in state 1.1");
          block();
          return false;
        }

      }

      private void op2(){
        System.out.println("\nAgent "+ getLocalName() + " in state 1.2
is waiting for a message");
        //Using a blocking receive causes the block
       // of all the behaviours
        ACLMessage msg = blockingReceive(5000);
        if(msg != null)
          System.out.println("\nAgent      "+      getLocalName()      +
             "  received  the  following  message  in  state  1.2:  "
             +msg.toString());
        else
          System.out.println("\nNo message received in state 1.2");


      }

      private void op3() {
        System.out.println("\nAgent: "+getLocalName()+
           " in state 1.3 is waiting for a message");
        MessageTemplate m1 =
           MessageTemplate.MatchPerformative(ACLMessage.INFORM);
        MessageTemplate m2 = MessageTemplate.MatchLanguage("PlainText");
        MessageTemplate m3 =
           MessageTemplate.MatchOntology("ReceiveTest");
        MessageTemplate m1andm2 = MessageTemplate.and(m1,m2);
        MessageTemplate m1andm2_and_m3 =
          MessageTemplate.and(m1andm2, m3);
        //blockingReceive and template
        ACLMessage msg = blockingReceive(m1andm2_and_m3);
        if (msg!= null)
          System.out.println("\nAgent      "+      getLocalName()      +
             "  received  the  following  message  in  state  1.3:  "
             + msg.toString());
```

```
        else
          System.out.println("\nNo message received in state 1.3");
        }
      } // End of my3StepBehaviour class


      protected void setup() {
        my3StepBehaviour mybehaviour = new my3StepBehaviour(this);
        addBehaviour(mybehaviour);
      }
    }
```

3.5 Interaction Protocols

FIPA specifies a set of standard interaction protocols, that can be used as standard templates to build agent conversations. For every conversation among agents, JADE distinguishes the *Initiator* role (the agent starting the conversation) and the *Responder* role (the agent engaging in a conversation after being contacted by some other agent). JADE provides ready made behaviour classes for both roles in conversations following most FIPA interaction protocols. These classes can be found in `jade.proto` package, as described in this section.

All Initiator behaviours terminate and are removed from the queue of the agent tasks, as soon as they reach any final state of the interaction protocol. In order to allow the re-use of the Java objects representing these behaviours without having to recreate new objects, all initiators include a number of `reset` methods with the appropriate arguments. Furthermore, all Initiator behaviours, but FipaRequestInitiatorBehaviour, are 1:N, i.e. can handle several responders at the same time.

All Responder behaviours, instead, are cyclic and they are rescheduled as soon as they reach any final state of the interaction protocol. Notice that this feature allows the programmer to limit the maximum number of responder behaviours that the agent should execute in parallel. For instance, the following code ensures that a maximum of two contract-net tasks will be executed simultaneously.

```
Protected void setup() {
   addBehaviour(new FipaContractNetResponderBehaviour(<arguments>));
   addBehaviour(new FipaContractNetResponderBehaviour(<arguments>));
}
```

A complete reference for these classes can be found in JADE HTML documentation and class reference.

Since JADE 2.4 a new couple of classes has been added, `AchieveREInitiator/Responder`, that provides an effective implementation for all the FIPA-Request-like interaction protocols, included FIPA-Request itself, FIPA-query, FIPA-propose, FIPA-Request-When, FIPA-recruiting, FIPA-brokering, FIPA-subscribe, ... **It is intention of the authors to keep only this couple of classes and soon deprecate the other jade.proto classes.**

3.5.1 AchieveRE (Achieve **R**ational **E**ffect)

The fundamental view of messages in FIPA ACL is that a message represents a communicative act, just one of the actions that an agent can perform. The FIPA standard specifies for each communicative act the Feasibility Preconditions (the conditions which need to be true

before an agent can execute the action, i.e. before the message can be sent) and the Rational Effect, i.e. the expected effect of the action or, in other terms, the reason why the message is sent. The standard specifies also that, having performed the act (i.e. having sent the message), the sender agent is not entitled to believe that the rational effect necessarily holds; for instance, given its autonomy, the receiver agent might simply decide to ignore the received message. That is not desirable in most applications because it generates an undesirable level of uncertainty. For this reason, instead of sending a single message, an interaction protocol should be initiated by the sender agent that allows to verify if the expected rational effect has been achieved or not.

FIPA has already specified a number of these interaction protocols, like FIPA-Request, FIPA-query, FIPA-propose, FIPA-Request-When, FIPA-recruiting, FIPA-brokering, FIPA-subscribe, that allows the initiator to verify if the expected rational effect of a single communicative act has been achieved. Because they share the same structure, JADE provides the `AchieveREInitiator/Responder` couple of classes which are a single homogeneous implementation of all these kind of interaction protocols.

Figure 5 shows the structure of these interaction protocols. The initiator sends a message (in general it performs a communicative act, as shown in the white box). The responder can then reply by sending a **not-understood**, or a **refuse** to achieve the rational effect of the communicative act, or also an **agree** message to communicate the agreement to perform (possibly in the future) the communicative act, as shown in the first row of shaded boxes. The responder performs the action and, finally, must respond with an **inform** of the result of the action (eventually just that the action has been done) or with a **failure** if anything went wrong. Notice that we have extended the protocol to make optional the transmission of the **agree** message. Infact, in most cases performing the action takes so short time that sending the **agree** message is just an useless and uneffective overhead; in such cases, the **agree** to perform the communicative act is subsumed by the reception of the following message in the protocol.

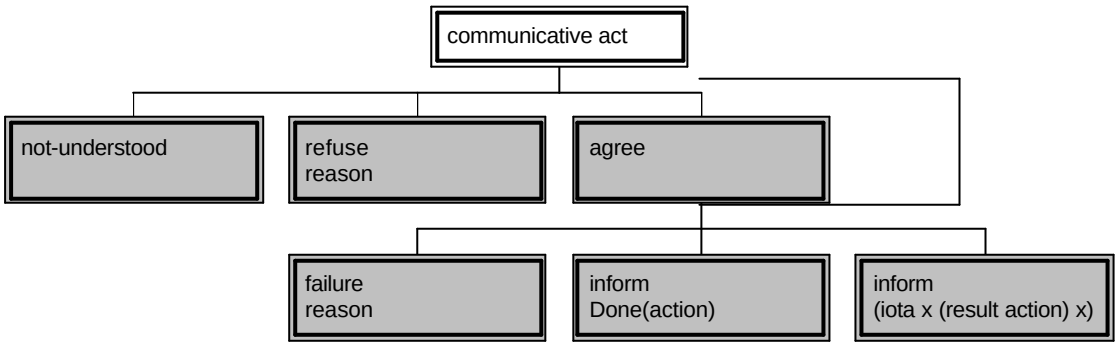

Figure 5 - Homogeneous structure of the interaction protocols.

3.5.1.1 AchieveREInitiator

An instance of this class can be easily constructed by passing, as argument of its constructor, the message used to initiate the protocol. It is important that this message has the right value for the *protocol* slot of the *ACLMessage* as defined by the constants in the interface *FIPAProtocolNames*.

Notice that this *ACLMessage* object might also be incomplete when the constructor of this class is created; the method *prepareRequests* can be overridden in order to return the complete *ACLMessage* or, more exactly (because this initiator allows to manage a 1:N conversation) a Vector of *ACLMessage* objects to be sent.

The class can be easily extended by overriding one (or all) of its *handle...* methods which provide hooks to handle all the states of the protocol. For instance the method *handleRefuse* is called when a *refuse* message is received.

Skilled programmers might find useful, instead of extending this class and overriding some of its methods, registering application-specific *Behaviours* as handler of the states of the protocol, including, for instance, another *AchieveREInitiator* behaviour to request a password before agreeing to perform the communicative act. The methods *registerHandle...* allow to do that. A mix of overridden methods and registered behaviours might often be the best solution.

It is worth clarifying the distinction between the following three handlers:

- *handleOutOfSequence* handles all the unexpected received messages which have the proper conversation-id or in-reply-to value
- *handleAllResponses* handles all the received first responses (i.e. *not-understood, refuse, agree)* and it is called after having called *handleNotUnderstood/Refuse/Agree* for each single response received. In case of 1:N conversations the override of this method might be more useful than the override of the other methods because this one allows to handle all the messages in a single call.
- *handleAllResultNotifications* handles all the received second responses (i.e. *failure, inform)* and it is called after having called *handleFailure/Inform* for each single response received. In case of 1:N conversations the override of this method might be more useful than the override of the other methods because this one allows to handle all the messages in a single call.

A set of variables **(they are not constants!)** is available (*..._KEY*) that provide the keys to retrieve the following information from the dataStore of this Behaviour:

- *getDataStore().get(ALL_RESPONSES_KEY)* returns a *Vector* of *ACLMessage* object with all the first responses (i.e. *not-understood, refuse, agree)*
- *getDataStore().get(ALL_RESULT_NOTIFICATIONS_KEY)* returns a *Vector* of *ACLMessage* object with all the second responses (i.e. *failure, inform)*
- *getDataStore().get(REQUEST_KEY)* returns the *ACLMessage* object passed in the constructor of the class
- *getDataStore().get(ALL_REQUESTS_KEY)* returns the *Vector* of *ACLMessage* objects returned by the *prepareRequests* method. **Remind that** if a Behaviour is registered as handler of the *PrepareRequests* state, it is responsibility of this behaviour to put into the datastore the proper *Vector* of *ACLMessage* objects (bound at the right key) to be sent by this initiator.

This implementation manages the expiration of the timeout, as expressed by the value of the *reply-by* slot of the sent *ACLMessage* objects. In case of 1:N conversation, the minimum is evaluated and used between the values of all the *reply-by* slot of the sent *ACLMessage* objects. Notice that, as defined by FIPA, this timeout refers to the time when the first response (e.g. the *agree* message) has to be received. If applications need to limit the timeout for receiving the last *inform* message, they must embed this limit into the content of the message by using application-specific ontologies.

3.5.1.2 AchieveREResponder

This class is the implementation of the responder role. It is very important to pass the right message template as argument of its constructor, in fact it is used to select which received *ACLMessage* should be served. The method *createMessageTemplate* can be used to create a message template for a given interaction protocol, but also more selective templates might be useful in some cases, for example to have an instance of this class for each possible sender agent.

The class can be easily extended by overriding one (or all) of its *prepare...* methods which provide hooks to handle the states of the protocol and, in particular, to prepare the response messages. The method *prepareResponse* is called when an initiator's message is received and the first response (e.g. the *agree)* must be sent back; the method *prepareResultNotification* is called, instead, when the rational effect must be achieved (for instance the action must be performed in case of a FIPa-Request protocol) and the final response message must be sent back (e.g. the *inform(done)*). **Take care** in returning the proper message and setting all the needed slots of the ACLMessage; in general it is highly recommended to create the reply message by using the method *createReply()* of the class *ACLMessage*.

Skilled programmers might find useful, instead of extending this class and overriding some of its methods, registering application-specific *Behaviours* as handler of the states of the protocol. The methods *registerPrepare...* allow to do that. A mix of overridden methods and registered behaviours might often be the best solution.

A set of variables **(they are not constants!)** is available (*..._KEY*) that provide the keys to retrieve the following information from the dataStore of this Behaviour:

- *getDataStore().get(REQUEST_KEY)* returns the *ACLMessage* object received by the initiator
- *getDataStore().get(RESPONSE_KEY)* returns the first *ACLMessage* object sent to the initiator
- *getDataStore().get(RESULT_NOTIFICATION_KEY)* returns the second *ACLMessage* object sent to the initiator

Remind that if a Behaviour is registered as handler of the *Prepare...* states, it is responsibility of this behaviour to put into the datastore (bound at the right key) the proper *ACLMessage* object to be sent by this responder.

3.5.1.3 Example of using these two generic classes for implementing a specific FIPA protocol

The two classes described above can easily be used for implementing the interaction protocols defined by FIPA.

The following example shows how to add a FIPA-Request initiator behaviour:

```
ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
request.setProtocol(FIPAProtocolNames.FIPA_REQUEST);
request.addReceiver(new AID("receiver", AID.ISLOCALNAME));
myAgent.addBehaviour( new AchieveREInitiator(myAgent, request) {
  protected void handleInform(ACLMessage inform) {
            System.out.println("Protocol finished. Rational Effect achieved.
Received the following message: "+inform);
      }
});
```

The following example shows instead how to add a FIPA-Request responder behaviour:

```
MessageTemplate mt =
      AchieveREResponder.createMessageTemplate(FIPAProtocolNames.FIPAREQUEST);
myAgent.addBehaviour( new AchieveREResponder(myAgent, mt) {
  protected ACLMessage prepareResultNotification(ACLMessage request, ACLMessage
response) {
      System.out.println("Responder has received the following message: " +
request);
      ACLMessage informDone = request.createReply();
      informDone.setPerformative(ACLMessage.INFORM);
```

```
        informDone.setContent("inform done");
        return informDone;
        }
});
```

3.5.2 FIPA-Contract-Net

This interaction protocol allows the Initiator to send a Call for Proposal to a set of responders, evaluate their proposals and then accept the preferred one (or even reject all of them). The interaction protocol is deeply described in the FIPA specifications while the following figure is just a simplification for the programmer.

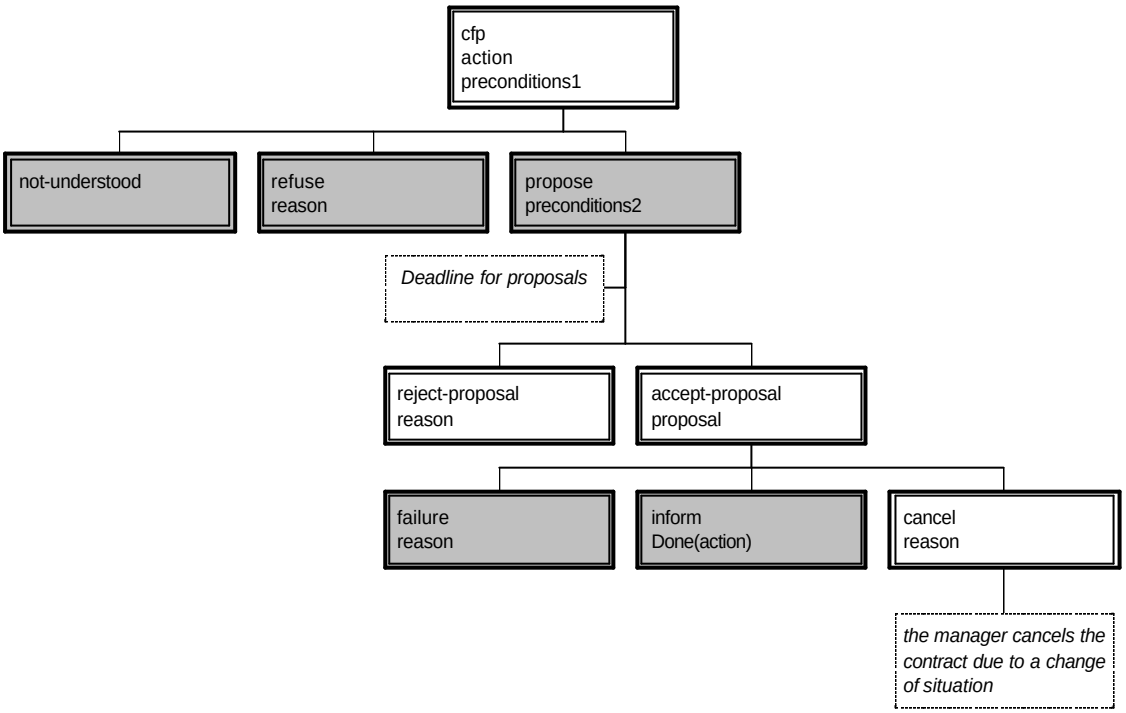

Figure 6 - FIPA-Contract-Net Interaction Protocol

3.5.2.1 FipaContractNetInitiatorBehaviour

This abstract behaviour implements the fipa-contract-net interaction protocol from the point of view of the agent initiating the protocol, that is the agent that sends the cfp (call for proposal) message.

The constructor of this behaviour takes 3 parameters
`public FipaContractNetInitiatorBehaviour(Agent a, ACLMessage msg, List responders)`
the calling agent, the CFP message to be sent and the group of agents to which the message should be sent. In fact, the protocol is implemented 1:N with one initiator and several responders.

The programmer should implement the two methods `handleProposeMessages` and `handleFinalMessages` to handle the two states of the protocol from the point of view of the initiator.

Under some circumstances, for instance when using the SL-0 content language, the content of the CFP message needs to be adapted to each receiver. For this reason, the method `createcfpcontent` is called before sending each message. The default implementation returns exactly the same content independently of the receiver; the programmer might also wish to override this default implementation.

The behaviour takes also care of handling timeouts in waiting for the answers. The timeout is got from the `reply-by` field of the ACLMessage passed in the constructor; if it was not set, then an infinite timeout is used. If the timeout expires without having received any answer, the method `handleXXXMessages` is executed by passing an empty vector of messages. Of course, late answers that arrive after the timeout expires are not consumed and remain in the private queue of incoming ACLmessages. Because this queue has a maximum size, these messages will be removed after the queue becomes full.

3.5.3 FipaContractNetResponderBehaviour

This abstract behaviour class implements the fipa-contract-net interaction protocol from the point of view of a responder to a call for proposal (cfp) message.

The programmer should extend this class by implementing the `handleXXX` methods that are called to handle the types of messages that can be received in this protocol.

3.5.4 Generic states of interaction protocols

The package jade.proto.states contains implementations for some generic states of interaction protocols which might be useful to register as handlers.

3.5.4.1 HandlerSelector class

This abstract class of the package jade.proto.states provides an implementation for a generic selector of handler, where an handler is a jade.core.behaviours.Behaviour.

The constructor of the class requires passing three arguments: a reference to the Agent, a reference to the DataStore where the selection variable can be retrieved, and, finally, the access key to retrieve the selection variable from this DataStore.

This selection variable will be later passed as argument to the method *getSelectionKey* that must return the key for selecting between the registered handlers. In fact, each handler must be registered with a key via the method *registerHandler*.

Useful examples of usage of this class are, for instance, the selection of a different handler for each action name (es. the action "register" is handled by the behaviour "registerBehaviour", the action modify by another one, and so on for each action). This class is generic enough to allow a large variety of selection systems, such as based on the message sender, the content language, the ontology, ... the programmer just needs to extend the class and override its method `getSelectionKey`

3.5.4.2 MsgReceiver class

This is a generic implementation for waiting for the arrival of a given message of the expiration of a given timeout. Refer to the javadoc for the documentation of its usage.

3.6 Application-defined content languages and ontologies

3.6.1 Rationale

When an agent A communicates with another agent B, a certain amount of information I is transferred from A to B by means of an ACL message.

Inside the ACL message I is represented as a content expression consistent with a proper content language (e.g. SL) and encoded in a proper format (e.g. string).

Both A and B have their own (possibly different) way of internally representing I.

Taking into account that the way an agent internally represents a piece information must allow an easy handling of that piece of information, it is quite clear that the representation used in an ACL content expression is not suitable for the inside of an agent.

For example the information that *the person Giovanni is 33 years old* in an ACL content expression could be represented as the string

```
(person (name Giovanni) (age 33) )
```

Storing this information inside an agent simply as a string variable is not suitable to handle the information as e.g. getting the age of Giovanni would require each time to parse the string.

Considering software agents written in Java (as JADE agents are), information can conveniently be represented inside an agent as Java objects.

For example representing the above information about Giovanni as an instance (an object) of an application-specific class

```
class Person {
  String name;
  int age;

  public String getName() {return name; }
  public void setName(String n) {name = n; }
  public int getAge() {return age; }
  public void setAge(int a) {age = a; }
  ….
}
```

initialized with

```
name = "Giovanni";
age = 33;
```

would allow to handle it very easily.

It is clear however that if on the one hand information handling inside an agent is eased, on the other hand each time agent A sends a piece of information I to agent B,

1) A needs to convert his internal representation of I into the corresponding ACL content expression representation and B needs to perform the opposite conversion.
2) Moreover B should also check that I complies with the rules (i.e. for instance that the age of Giovanni is actually an integer value) of the ontology by means of which both A and B ascribe a proper meaning to I.

The support for application-defined ontology and content languages provided by JADE is designed to support agent internal representation of information as Java objects, as described above, by minimizing the developer effort in performing the above conversion and check operations.

3.6.2 The conversion pipeline

Each time an information has to be inserted into or extracted from an ACL content expression the JADE framework automatically performs the pipeline depicted in figure.

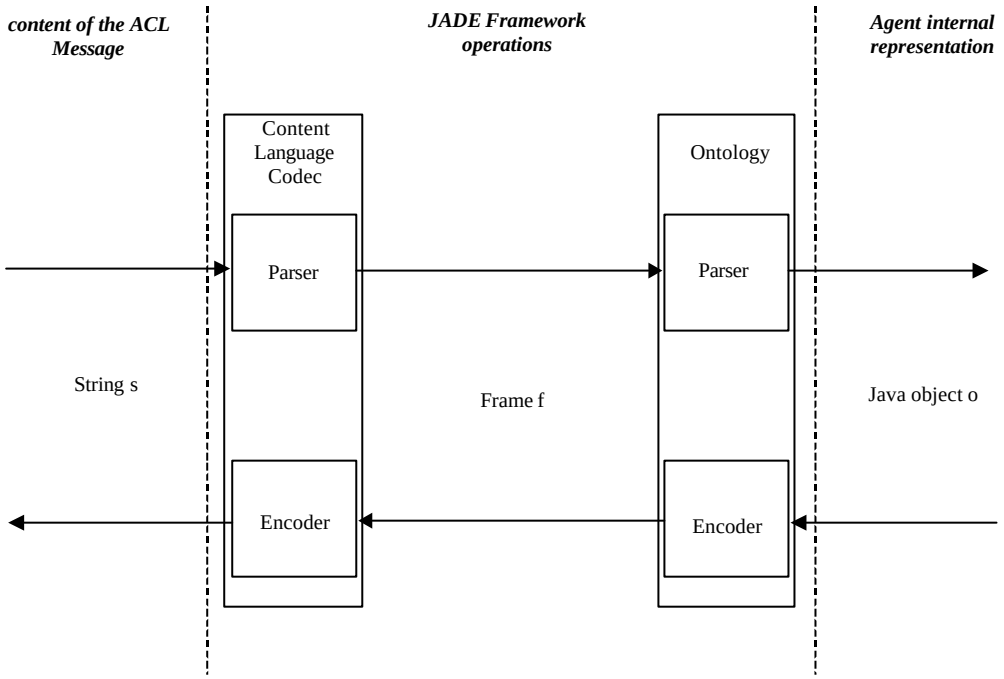

Figure 7 - Pipeline of the message content encoding/decoding

First an appropriate content language **codec** object is able to parse a content expression and to convert it into a t-uple[7] of `Frame`[8] objects.

An appropriate **ontology** object is then able to check whether a `Frame` object is consistent with one of the schemas defining the **roles**[9] included in the ontology and, in this case, to convert

[7] A content expression can include more than one entity in the domain of discourse. E.g. the content of a REFUSE ACL message is a t-uple with 2 elements: an action expression and the reason why the agent sending the message refuses to accomplish that action.

[8] The JADE Framework uses an internal representation of information based on the class `Frame`. A `Frame` object has a name and a set of slots each one being characterized by a name, a position (within the frame) and an untyped value. Whatever entity in the domain of discourse (i.e. whatever information) can be represented as a `Frame` object.

[9] An ontology basically includes all the concepts, predicates and actions, collectively called **roles**, that are meaningful for the agents sharing this ontology. For instance the concepts *Company* and *Person*, the predicate *WorfsFor* and the action *Engage* can be **roles** in an ontology dealing with employees. All elements in the domain of discourse (e.g. the person Giovanni) are instances of one of the **roles** composing the ontology.

the `Frame` object into a properly initialized instance of the application-specific class (e.g. the Person class mentioned above) representing the matched role.

The opposite pipeline allows to convert a sentence belonging to the domain of discourse, and represented as a Java object, into the appropriate content language and encoding.

The JADE framework hides the stages of this pipeline to the programmer who just needs to call the following methods of the `Agent` class.

```
List extractContent(ACLMessage msg);
void fillContent(ACLMessage msg, List content);
```

As already mentioned the content of an ACL message is in general a t-uple of entity inn the domain of discourse. In Java this is represented as a `List`.

The programmer however has to create and add to the resources of the agent the codec and ontology objects mentioned above as described in the followings.

3.6.3 Codec of a Content Language

Each content language codec in JADE must implement the interface `jade.lang.Codec` and, in particular, the two methods `decode`() and `encode`() to respectively

- parse the content in an ACL message and convert it into a `List` of `Frame` objects.
- encode the content from a `List` of `Frame` objects into the content language syntax and encoding.

The `Frame` class is a neutral type (i.e. it does not distinguish between concepts, actions and predicates), that has been designed in order to allow accessing its slots both by name (e.g. *(divide :dividend 10 :divisor 2)*) and by position (e.g. *(divide 10 2)*).

This `Codec` object must then be added to the resources of each agent, which wishes to use that language, by using the method `registerLanguage`() available in the `Agent` class.

By means of this operation a `Codec` object is associated to a content language name. When the `fillContent`() and `extractContent`() methods are called the `Codec` object associated to the content language indicated in the `:language` slot of the ACL message will be used to perform the conversion pipeline described in previous chapter.

Notice that JADE already includes the `Codec` for SL-0 (one of the standard content languages defined by FIPA) that is the class `jade.lang.sl.SL0Codec`. For an agent using SL0 it will be therefore sufficient to insert the instruction

```
registerLanguage("SL0", new SL0Codec());
```

3.6.4 Creating an Ontology

Each ontology in JADE must implement the `jade.onto.Ontology` interface.

It is important to note however that in the adopted approach an ontology is represented by an instance of a class implementing the `jade.onto.Ontology` interface and not just by that class.

More in detail a class implementing the `jade.onto.Ontology` interface only embeds the definition of the semantic checks that will be performed when some information is received. For example an implementation can check that the age of a person is an integer value, while another implementation can also check that that integer value is > 0. All the ontological roles included in the ontology (such as the concept of person) must on the other hand be added at run-time to an instance of the above class.

Two instances `o1` and `o2` of the same class `O` implementing the `jade.onto.Ontology` interface can represent two different ontologies provided that at run-time different ontological roles are added to `o1` and `o2`.

A class, `jade.onto.DefaultOntology`, providing a default implementation of the `Ontology` interface is already provided by JADE. This is simple but still expected to be useful in most practical applications.

Creating an ontology requires the following steps:

- Defining an application-specific class for each role in the ontology
- Creating an object of class `DefaultOntology`
- Adding to that object all the ontological roles as described below.

Each ontological role is described by a name and a number of **slots**. The `SlotDescriptor` class is provided to describe the characteristics of a slot of an ontological role.

The method `addRole()` by means of which a role is added to an ontology object takes therefore the following parameters:

- A `String` indicating the name of the added role. This parameter is missing for an unnamed slot.
- An array of `SlotDescriptor` each one describing a slot of the added role.
- The Java class, if any, that represents the role.

For example, adding the *person* role described by the `Person` class mentioned above, to a previously created ontology object `myOnto` will look like

```
Ontology myOnto = new DefaultOntology();
……
myOnto.addRole(
  “Person”,
  new SlotDescriptor[]{
        new SlotDescriptor(“name”, Ontology.PRIMITIVE_SLOT,
        Ontology.STRING_TYPE, Ontology.M),
        new SlotDescriptor(“age”, Ontology.PRIMITIVE_SLOT,
        Ontology.INTEGER_TYPE,Ontology.O)
  },
  Person.class
);
```

Each slot has

- A **name** and/or a **position** (implicitly defined by the position in the array of `SlotDescriptors`) identifying the slot.
- A **category** stating that the value of the slot can be a primitive entity such as a string or an integer (`Ontology.PRIMITIVE_SLOT`), an instance of another ontological role (`Ontology.FRAME_SLOT`) or a set (`Ontology.SET_SLOT`) or sequence (`Ontology.SEQUENCE_SLOT`) of entities.
- A **type** defining the primitive type (for primitive slots) or role (for frame slots) of the value of the slot or of the elements in the set/sequence in case of set slots or sequence slots.
- A **presence** flag defining whether the slot is mandatory (`Ontology.M`) or optional (`Ontology.O`).

In the above case the *person* role has two named slots called *name* and *age*. The first is mandatory (an exception will be thrown if this slot has a null value) and permitted values are of type String. The second is optional and permitted values are of type Integer.

As a further example three other roles are added to the ontology represented by the `myOnto` object.

- *Address*, with three named slots, *street*, *number* and *city*, of type `String`, `Integer` and `String` respectively and all mandatory.
- *Company* with two named slots, *name* and *address*, of type `String` and *Address* (i.e. the values of this slot are instances of the *Address* role) respectively, one mandatory and the other optional.
- *Engage* (the action of engaging a person in a company) with two unnamed slots of type *Person* and *Company* respectively and both mandatory.

Application specific class representing the *Address* role

```
public class Address {
  private String street;
  private Integer number;
  private String city;
  public String getStreet() { return string; }
  public void setStreet(String s) { string = s; }
  public Integer getNumber() { return number; }
  public void setNumber(Integer n) { number = n; }
  public String getCity() { return city; }
  public void setCity(String c) { city  = c; }
}
```

Application specific class representing the *Company* role

```
public class Company {
    private String name;
    private Address  address;

    public void   setName(String n) { name = n; }
    public String getName() { return name; }
    public void   setAddress(Address a) { address = a; }
    public Address  getAddress() { return address; }
}
```

Application specific class representing the *Engage* role

```
public class Engage {
    private Person personToEngage;
    private Company engager;

    public void   set_0(Person p) { personToEngage = p; }
    public Person  get_0() { return personToEngage; }
    public void   set_1(Company c) { engager = c; }
    public Company  get_1() { return engager; }
}
```

Code for adding the *Address*, *Company* and *Engage* roles to the ontology.

```
myOnto.addRole(
  "Address",
  new SlotDescriptor[]{
       new SlotDescriptor("street", Ontology.PRIMITIVE_SLOT,
       Ontology.STRING_TYPE, Ontology.M),
       new SlotDescriptor("number", Ontology.PRIMITIVE_SLOT,
       Ontology.INTEGER_TYPE,Ontology.M)
       new SlotDescriptor("city", Ontology.PRIMITIVE_SLOT,
       Ontology.SRING_TYPE,Ontology.M)
  },
  Address.class
);

myOnto.addRole(
  "Company",
  new SlotDescriptor[]{
       new SlotDescriptor("name", Ontology.PRIMITIVE_SLOT,
       Ontology.STRING_TYPE, Ontology.M),
       new SlotDescriptor("address", Ontology.FRAME_SLOT,
       "Address" ,Ontology.O)
  },
  Company.class
);

myOnto.addRole(
  "engage",
  new SlotDescriptor[]{
       new SlotDescriptor(Ontology.FRAME_SLOT, "Person"
       Ontology.M),
       new SlotDescriptor(Ontology.FRAME_SLOT, "Company",
       Ontology.M)
  },
  Engage.class;
```

```
);
```

The ontology object must finally be added to the resources of each agent wishing to use it, by using the method `registerOntology()` available in the `Agent` class.

By means of this operation an `Ontology` object is associated to a name. When the `fillContent()` and `extractContent()` methods are called the `Ontology` object associated to the content language indicated in the `:ontology` slot of the ACL message will be used to perform the conversion pipeline described in section 3.6.2.

3.6.5 Application specific classes representing ontological roles

In order to represent an ontological role (i.e. in order to be accepted by the `Ontology` object), a Java class must obey to some rules:

1) For each slot in the represented role named *XXX*, of category `Ontology.PRIMITIVE_SLOT` or `Ontology.FRAME_SLOT` and of type `T` the class must have two accessible methods with the following signature:

```
public T getXXX();
public void setXXX(T t);
```

2) For each slot in the represented role named *XXX*, of category `Ontology.SET_SLOT` or `Ontology.SEQUENCE_SLOT` and with elements of type `T`, the class must have two accessible methods with the following signature:

```
public Iterator getAllXXX();
public void addXXX(T t);
```

3) For each unnamed slot use "_p" (being p the position of the slot) instead of the slot name for the get and set methods (see the Engage class mentioned above for an example).

4) In all previous cases the type `T` cannot be a primitive type such as `int`, `float` or `boolean`. Use `Integer`, `Float`, `Boolean` …. instead.

3.6.6 Discovering the ontological role of a Java object representing an entity in the domain of discourse

As already mentioned, when an ACL message is received, provided that the proper ontology and content language codec objects has been previously registered, the content of the ACL message can be easily converted into a list of proper Java objects by means of the `extractContent()` method .

```
List l = extractContent( msg );
```

In general however the receiving agent does not know a-priori the role of each Java object in the list. In order to discover it the ontology object must be used as described in the example below refering to the first object in the list.

```
Object obj = l.get(0);
Ontology onto = lookupOntology(msg.getOntology());
String roleName = onto.getRoleName(obj.getClass());
```

The `lookupOntology`() is a method of the `Agent` class that returns the ontology object previously associated to a given name by calling the `registerOntology`() method.

Once discovered the role of the entity represented by an object it will be possible to cast it to the application specific class representing that role.

3.6.7 Setting and getting the content of an ACL message.

Having registered a content language codec and an ontology with the agent, it is possible to exploit the automatic support of the JADE framework to set and get the content of an ACL message. The `Agent` class provides two methods for this purpose: `extractContent()` and `fillContent()` to implement parsing and encoding operations on the message content, respectively.

The first method extracts the content from an ACL message and returns a `List` of Java objects (one object for each element of the tuple in the content) by calling the appropriate content language `Codec` (according to the value of the *:language* parameter of the ACL message) and the appropriate Ontology (according to the value of the *:ontology* parameter of the ACL message).

The second method, instead, makes the opposite operation, that is it fills in the content of an ACL message by interpreting a `List` of Java objects with the appropriate `Ontology` and content language `Codec`, as specified by the values of the *:ontology* and the *:language* parameter of the ACL message.

Refer to the javadoc documentation for a detailed description of the usage of these two methods.

3.7 Support for Agent Mobility

Using JADE, application developers can build mobile agents, which are able to migrate or copy themselves across multiple network hosts. In this version of JADE, only *intra-platform* mobility is supported, that is a JADE mobile agent can navigate across different agent containers but it is confined to a single JADE platform.

Moving or cloning is considered a state transition in the life cycle of the agent. Just like all the other life cycle operation, agent motion or cloning can be initiated either by the agent itself or by the AMS. The `Agent` class provides a suitable API, whereas the AMS agent can be accessed via FIPA ACL as usual.

Mobile agents need to be *location aware* in order to decide when and where to move. Therefore, JADE provides a proprietary ontology, named *jade-mobility-ontology*, holding the necessary concepts and actions.

This ontology is contained within the `jade.domain.MobilityOntology` class, and it is an example of the new application-defined ontology support.

3.7.1 JADE API for agent mobility.

The two public methods `doMove()` and `doClone()` of the `Agent` class allow a JADE agent to migrate elsewhere or to spawn a remote copy of itself under a different name. Method `doMove()` takes a `jade.core.Location` as its single parameter, which represents the intended destination for the migrating agent. Method `doClone()` also takes a `jade.core.Location` as parameter, but adds a `String` containing the name of the new agent that will be created as a copy of the current one.

Looking at the documentation, one finds that `jade.core.Location` is an abstract interface, so application agents are not allowed to create their own locations. Instead, they must ask the AMS for the list of the available locations and choose one. Alternatively, a JADE agent can also request the AMS to tell where (at which location) another agent lives.

Moving an agent involves sending its code and state through a network channel, so user defined mobile agents must manage the serialization and unserialization process. Some among the various resources used by the mobile agent will be moved along, while some others will be disconnected before moving and reconnected at the destination (this is the same distinction between `transient` and non-`transient` fields used in the *Java Serialization API*). JADE makes available a couple of matching methods in the `Agent` class for resource management.

For agent migration, the `beforeMove()` method is called at the starting location just before sending the agent through the network (with the scheduler of behaviours already stopped), whereas the `afterMove()` method is called at the destination location as soon as the agent has arrived and its identity is in place (but the scheduler has not restarted yet).

For agent cloning, JADE supports a corresponding method pair, the `beforeClone()` and `afterClone()` methods, called in the same fashion as the `beforeMove()` and `afterMove()` above. The four methods above are all `protected` methods of the `Agent` class, defined as empty placeholders. User defined mobile agents will override the four methods as needed.

3.7.2 JADE Mobility Ontology.

The *jade-mobility-ontology* ontology contains all the concepts and actions needed to support agent mobility. JADE provides the class `jade.domain.MobilityOntology`, working as a *Singleton* and giving access to a single, shared instance of the JADE mobility ontology through the `instance()` method.

The ontology contains ten frames (six concepts and four actions), and a suitable inner class is associated with each frame using a `RoleEntityFactory` object (see Section 3.6.4 for details). The following list shows all the frames and their structure.

- `Mobile-agent-description`; describes a mobile agent going somewhere. It is represented by the `MobilityOntology.MobileAgentDescription` inner class.

Slot Name	Slot Type	Mandatory/Optional
name	**AID**	**Mandatory**
destination	**Location**	**Mandatory**
agent-profile	mobile-agent-profile	Optional
agent-version	String	Optional
signature	String	Optional

- `mobile-agent-profile`; describes the computing environment needed by the mobile agent. It is represented by the `MobilityOntology.MobileAgentProfile` inner class.

Slot Name	Slot Type	Mandatory/Optional
system	mobile-agent-system	Optional
language	mobile-agent-language	Optional
os	**Mobile-agent-os**	**Mandatory**

- `mobile-agent-system`; describes the runtime system used by the mobile agent. It is represented by the `MobilityOntology.MobileAgentSystem` inner class.

Slot Name	Slot Type	Mandatory/Optional
name	**String**	**Mandatory**
major-version	**Long**	**Mandatory**
minor-version	Long	Optional
dependencies	String	Optional

- `mobile-agent-language`; describes the programming language used by the mobile agent. It is represented by the `MobilityOntology.MobileAgentLanguage` inner class.

Slot Name	Slot Type	Mandatory/Optional
name	**String**	**Mandatory**
major-version	**Long**	**Mandatory**
minor-version	Long	Optional
dependencies	String	Optional

- `mobile-agent-os`; describes the operating system needed by the mobile agent. It is represented by the `MobilityOntology.MobileAgentOS` inner class.

Slot Name	Slot Type	Mandatory/Optional
name	**String**	**Mandatory**
major-version	**Long**	**Mandatory**
minor-version	Long	Optional
dependencies	String	Optional

- `Location`; describes a location where an agent can go. It is represented by the `MobilityOntology.Location` inner class.

Slot Name	Slot Type	Mandatory/Optional
name	**String**	**Mandatory**
protocol	**String**	**Mandatory**

address	String	Mandatory

- `move-agent`; the action of moving an agent from a location to another. It is represented by the `MobilityOntology.MoveAction` inner class.

  This action has a single, unnamed slot of type `mobile-agent-description`. The argument is mandatory.

- `clone-agent`; the action performing a copy of an agent, possibly running on another location. It is represented by the `MobilityOntology.CloneAction` inner class.

  This action has two unnamed slots: the first one is of `mobile-agent-description` type and the second one is of `String` type. Both arguments are mandatory.

- `where-is-agent`; the action of requesting the location where a given agent is running. It is represented by the `MobilityOntology.WhereIsAgent` inner class.

  This action has a single, unnamed slot of type `AID`. The argument is mandatory.

- `query-platform-locations`; the action of requesting the list of all the platform locations. It is represented by the `MobilityOntology.QueryPlatformLocations` inner class.

This action has no slots.

Notice that this ontology has no counter-part in any FIPA specifications. It is intention of the JADE team to update the ontology as soon as a suitable FIPA specification will be available.

3.7.3 Accessing the AMS for agent mobility.

The JADE AMS has some extensions that support the agent mobility, and it is capable of performing all the four actions present in the *jade-mobility-ontology*. Every mobility related action can be requested to the AMS through a *FIPA-request* protocol, with *jade-mobility-ontology* as `ontology` value and *FIPA-SL0* as `language` value.

The `move-agent` action takes a `mobile-agent-description` as its parameter. This action moves the agent identified by the `name` and `address` slots of the `mobile-agent-description` to the location present in the `destination` slot.

For example, if an agent wants to move the agent *Peter* to the location called *Front-End*, it must send to the AMS the following ACL `request` message:

```
(REQUEST
 :sender (agent-identifier :name RMA@Zadig:1099/JADE)
 :receiver (set (agent-identifier :name ams@Zadig:1099/JADE))
 :content  (
   (action (agent-identifier :name ams@Zadig:1099/JADE)
           (move-agent (mobile-agent-description
```

```
              :name (agent-identifier :name Johnny@Zadig:1099/JADE)
              :destination (location
                :name Main-Container
                :protocol JADE-IPMT
                :address Zadig:1099/JADE.Main-Container )
              )
             )
    )
  )
  :reply-with  Req976983289310
  :language  FIPA-SL0
  :ontology  jade-mobility-ontology
  :protocol  fipa-request
  :conversation-id  Req976983289310
 )
```

The above message was captured using the JADE sniffer, using the *MobileAgent* example and the RMA support for moving and cloning agents.

Using JADE ontology support, an agent can easily add mobility to its capabilities, without having to compose ACL messages by hand.

First of all, the agent has to create a new `MobilityOntology.MoveAction` object, fill its argument with a suitable `MobilityOntology.MobileAgentDescription` object, filled in turn with the name and address of the agent to move (either itself or another mobile agent) and with the `MobilityOntology.Location` object for the destination. Then, a single call to the `Agent.fillContent()` method can turn the `MoveAction` Java object into a `String` and write it into the `content` slot of a suitable `request` ACL message.

The `clone-agent` action works in the same way, but has an additional `String` argument to hold the name of the new agent resulting from the cloning process.

The `where-is-agent` action has a single `AID` argument, holding the identifier of the agent to locate. This action has a result, namely the location for the agent, that is put into the `content` slot of the `inform` ACL message that successfully closes the protocol.

For example, the `request` message to ask for the location where the agent *Peter* resides would be:

```
 (REQUEST
   :sender   (agent-identifier :name da1@Zadig:1099/JADE)
   :receiver (set (agent-identifier :name ams@Zadig:1099/JADE))
   :content  (( action
     (agent-identifier :name ams@Zadig:1099/JADE)
     (where-is-agent (agent-identifier :name Peter@Zadig:1099/JADE))
   ))
   :language  FIPA-SL0
   :ontology  jade-mobility-ontology
   :protocol  fipa-request
 )
```

The resulting `Location` would be contained within an `inform` message like the following:

```
(INFORM
  :sender (agent-identifier :name ams@Zadig:1099/JADE)
  :receiver  (set (agent-identifier :name da1@Zadig:1099/JADE))
  :content  ((result
    (action
      (agent-identifier :name ams@Zadig:1099/JADE)
      (where-is-agent (agent-identifier :name Peter@Zadig:1099/JADE))
    )
    (set (location
      :name Container-1
      :protocol JADE-IPMT
      :address Zadig:1099/JADE.Container-1
    ))
  ))
  :reply-with  da1@Zadig:1099/JADE976984777740
  :language  FIPA-SL0
  :ontology  jade-mobility-ontology
  :protocol  fipa-request
)
```

The `query-platform-locations` action takes no arguments, but its result is a set of all the `Location` objects available in the current JADE platform. The message for this action is very simple:

```
( REQUEST
 :sender (agent-identifier :name Johnny)
 :receiver (set (Agent-Identifier :name AMS))
 :content (( action (agent-identifier :name AMS)
            ( query-platform-locations ) ))
 :language FIPA-SL0
 :ontology jade-mobility-ontology
 :protocol fipa-request
)
```

If the current platform had three containers, the AMS would send back the following `inform` message:

```
( INFORM
 :sender (Agent-Identifier :name AMS)
 :receiver (set (Agent-Identifier :name Johnny))
 :content (( Result ( action (agent-identifier :name AMS)
                             ( query-platform-locations ) )
            (set (Location
                  :name Container-1
                  :transport-protocol JADE-IPMT
```

```
                    :transport-address IOR:000….Container-1 )
                  (Location
                    :name Container-2
                    :protocol JADE-IPMT
                    :address IOR:000….Container-2 )
                 (Location
                    :name Container-3
                    :protocol JADE-IPMT
                    :address IOR:000….Container-3 )
            )))
   :language  FIPA-SL0
   :ontology  jade-mobility-ontology
   :protocol  fipa-request
  )
```

The `MobilityOntology.Location` class implements `jade.core.Location` interface, so that it can be passed to `Agent.doMove()` and `Agent.doClone()` methods. A typical behaviour pattern for a JADE mobile agent will be to ask the AMS for locations (either the complete list or through one or more `where-is-agent` actions); then the agent will be able to decide if, where and when to migrate.

3.8 Using JADE from external Java applications

Since JADE 2.3, an in-process interface has been implemented that allows external Java applications to use JADE as a kind of library and to launch the JADE Runtime from within the application itself.

A singleton instance of the JADE Runtime can be obtained via the static method *jade.core.Runtime.instance()* Then, it provides two methods to create a JADE main-container or a JADE remote container (i.e. a container that joins to an existing main-container forming in this way a distributed agent platform); both methods requires passing as a parameter an object that implements the *jade.core.Profile* interface that can be queried, via the *getParameter(name)* method, to get the hostname and port number of the main container.

Both these two methods of the Runtime return a wrapper object, belonging to the package *jade.wrapper*, that wraps the higher-level functionality of the agent containers, such as installing and uninstalling MTPs (Message Transport Protocol)[10], killing the container (where just the container is killed while the external application remains alive) and, of course, creating new agents. The *createAgent* method of this container wrapper returns as well a wrapper object, which wraps some functionalities of the agent, but still tends to preserve the autonomy of agents. In particular, the application can control the life-cycle of the Agent but it cannot obtain a direct reference to the Agent object and, as a direct consequence, it cannot perform method calls on that object. **Notice that**, having created the agent, it still needs to be started via the method *start()*

The following code lists a very simple way to launch an agent from within an external applications (refer also to the *inprocess* directory in the JADE *examples* that contains an example of usage of this wrapping and in-process interface).

[10] see also the Administrator's Guide for this functionality

```
import jade.core.Runtime;
import jade.core.Profile;
import jade.core.ProfileImpl;
import jade.wrapper.*;
...
// Get a hold on JADE runtime
Runtime rt = Runtime.instance();
// Create a default profile
Profile p = new ProfileImpl();
// Create a new non-main container, connecting to the default
// main container (i.e. on this host, port 1099)
AgentContainer ac = rt.createAgentContainer(p);
// Create a new agent, a DummyAgent
// and pass it a reference to an Object
Object reference = new Object();
Object args[] = new Object[1];
args[0]=reference;
Agent dummy = ac.createAgent("inProcess",
              "jade.tools.DummyAgent.DummyAgent", reference);
// Fire up the agent
dummy.start();
...
```

Notice that this mechanism allows several different configurations for a JADE platform, such as a complete in-process platform composed of several containers on the same JVM, a platform partly in-process (i.e. containers launched by an external Java application) and partly out-of-process (i.e. containers launched from the command line).

4 A SAMPLE AGENT SYSTEM

We are presenting an example of an agent system explaining how to use the features available in JADE framework. In particular we will show the possibility of organising the behaviour of a single agent in different sub-behaviours and how the message exchange among agents takes place.

The agent system, in the example, is made of two agents communicating through FIPA request protocol.

This section is still to do. Please refer to JADE examples present in src/examples directory. Refer also to the `README` *file in* `src/examples` *directory to get some explanations of each example program.*

5 APPENDIX A: CONTENT-LANGUAGE INDEPENDENT API
FEDERICO BERGENTI (UNIVERSITY OF PARMA)

Application-specific ontologies describe the elements that agents use to create the content of messages, e.g., application-specific predicates and actions. The package `jade.content` (and its sub-packages) allows to create application-specific ontologies and to use them independently of the adopted content language: the code that implements the ontology and the code that sends and receives messages do not depend on the content language. The following is a description of such a package that uses `src/example/content` as a running example.

5.1 Creating an Application-Specific Ontology

An ontology defines a vocabulary and a set of relationships between the elements of the vocabulary. The relationships can be:

1) structural, e.g., the predicate `fatherOf` is defined over two parameters, a father and a set of children because we want to use it to say `fatherOf(John, (Mary, Lisa))`;
2) semantic, e.g., a concept belonging to the class `Man` also belongs to the class `Person`.

An application-specific ontology is implemented through one object of class `FullOntology` and it is characterized by:

1) one name;
2) one base ontology at most, i.e., an ontology that it extends;
3) a vocabulary;
4) a set of element schemata.

The following code implements the `People` ontology: it defines a constant for each element (concept, action, predicate, etc.) that we want to include in the vocabulary.

```
public class PeopleOntology extends FullOntology {
  // The name of this ontology.
  public static final String ONTOLOGY_NAME = "PEOPLE_ONTOLOGY";

  // Concepts, i.e., objects of the world.
  public static final String PERSON  = "PERSON";
  public static final String MAN     = "MAN";
  public static final String WOMAN   = "WOMAN";
  public static final String ADDRESS = "ADDRESS";

  // Slots of concepts, i.e., attributes of objects.
  public static final String NAME   = "NAME";
  public static final String STREET = "STREET";
  public static final String NUMBER = "NUMBER";
  public static final String CITY   = "CITY";

  // Predicates
  public static final String FATHER_OF = "FATHER_OF";
  public static final String MOTHER_OF = "MOTHER_OF";
```

```
  // Roles in predicates, i.e., names of arguments for predicates
  public static final String FATHER   = "FATHER";
  public static final String MOTHER   = "MOTHER";
  public static final String CHILDREN = "CHILDREN";

  // Actions
  public static final String MARRY = "MARRY";

  // Arguments in actions
  public static final String HUSBAND = "HUSBAND";
  public static final String WIFE    = "WIFE";

  private static PeopleOntology theInstance = new PeopleOntology();

  public static PeopleOntology getInstance() {
    return theInstance;
  }

  public PeopleOntology(Ontology base) {
    super(ONTOLOGY_NAME, ACLOntology.getInstance());

    // Add definitions of schemata here.
    …
  }
}
```

The constructor calls `super()` to assign a name to the ontology and to declare that it extends the `ACLOntology`. `People` ontology, and reasonably all ontologies, extends `jade.content.onto.ACLOntology` because we want to use in our messages the elements of such an ontology, e.g., variables and the `Done` predicate. If you do not need ACL concepts, you can extend `jade.content.onto.BasicOntology` in order to have only basic types, i.e., lists, strings and numbers. `ACLOntology` extends the `BasicOntology`.

The definition of the ontology in the example is not complete, we have to substitute dots with the definition of the element schemata. Element schemata are objects describing the structure of concepts, actions, predicate, etc. that we allow in our messages. In the `People` ontology they describe what a person is, what an address is, what a father is, etc. The following is the element schema for the concept of `Person`. This schema states that a `Person` is characterized by a `name` and an `address`:

```
// Get the element schema for strings from BasicOntology
PrimitiveSchema stringSchema =
  (PrimitiveSchema)getSchema(BasicOntology.STRING);

// Define the concept of Person
ConceptSchema personSchema = new ConceptSchema(PERSON);
personSchema.add(NAME,    stringSchema);
```

```
personSchema.add(ADDRESS, addressSchema, ObjectSchema.OPTIONAL);

// Add the schema to the ontology
add(personSchema);
```

PERSON, NAME and ADDRESS are string defined in the vocabulary and addressSchema has been defined before (not shown). Schemata that describe concepts support inheritance[11]. You can define the concept of Man as a refinement of the concept of Person:

```
ConceptSchema manSchema = new ConceptSchema(MAN);
manSchema.addSuperSchema(personSchema);
```

Element schema describe, in some way, the structure of a class of objects and therefore they can be associated with Java classes. This maps elements of the ontology that comply with a schema with Java objects of that class. The following is a class that might be associated with the Person concept:

```
public class Person extends Concept {
  private String  name    = null;
  private Address address = null;

  public void setName(String name) {
    this.name = name;
  }

  public void setAddress(Address address) {
    this.address = address;
  }

  public String getName() {
    return name;
  }

  public Address getAddress() {
    return address;
  }
}
```

The association between the class and the schema is performed when registering the concept in the ontology using the following statement:

[11] Only concept schemata support inheritance, all other schemata, e.g., predicate schemata, action schemata, etc do not.

```
// Add the schema to the ontology
addElement(personSchema, Person.class);
```

Associating classes with schemata is not mandatory, but helps because it support easier APIs, as shown later. In order to associate a class with a schema, the class must:

1) extend a class in `jade.content`, e.g., `Person` extends `Concept` because we want to associate it with a concept schema;
2) provide public `get`/`set` methods for each attribute (you can use basic types like `int` or `boolean`);
3) provide a constructor with no parameters, i.e., the default contructor.

Defining actions, predicates, etc. is just like defining concepts. Figure 8 shows the classes that corresponds to the elements we can use in our ontologies.

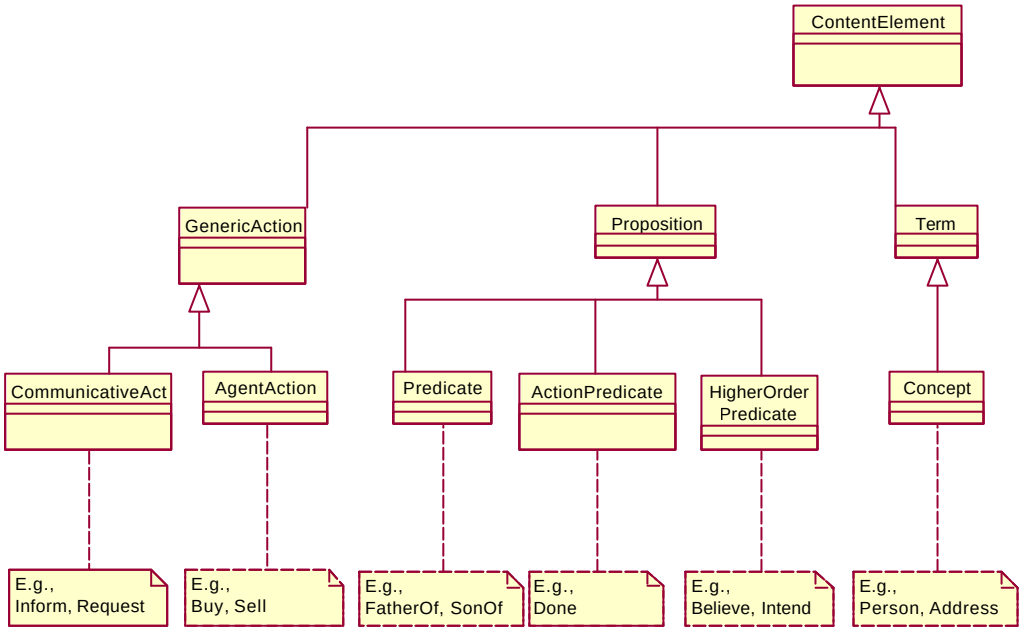

Figure 8 - Classes that correspond to elements of ontologies.

The following is the definition of the predicate `fatherOf`:

```
// Define a schema for the set of children
AggregateSchema childrenSchema = new AggregateSchema(BasicOntology.SET);

// Define the schema for fatherOf predicate
PredicateSchema fatherOfSchema = new PredicateSchema(FATHER_OF);

fatherOfSchema.add(FATHER,   manSchema);
fatherOfSchema.add(CHILDREN, childrenSchema);
```

```
// Add the predicate to the ontology
add(fatherOfSchema, FatherOf.class);
```

First we define a new schema to describe the set of children. Then we define the predicate schema for `fatherOf` by introducing two roles: the `father` and the `children`. Finally we register `fatherOfSchema` with the ontology associating it with the following class:

```
public class FatherOf extends Predicate {
  private List children = null;
  private Man  father   = null;

  public void setChildren(List children) {
    this.children = children;
  }

  public void setFather(Man father) {
    this.father = father;
  }

  public Man getFather() {
    return father;
  }

  public List getChildren() {
    return children;
  }
}
```

Note that we use `jade.util.leap.List` where the schema declares an aggregate, i.e., the `BasicOntology` associates any aggregate with `List`.

5.2 Sending and Receiving Messages

We restrict the description of ontologies to the features that support inter-agent communication. Other models, e.g., DAML+OIL, use description logics to provide richer description that support reasoning about concepts, predicates, actions, etc. In order to send and receive messages, we need *(i)* an ontology to provides the vocabulary and *(ii)* a codec (coder/encoder) to handle the syntax of the content language. These are registered with JADE through the *content manager*. The content manager provides methods for encoding and decoding the content of messages exploiting the registered ontologies and codecs. The following code registers the `People` ontology with the content manager and it also registers a codec called `jade.content.lang.j.JCodec`.

```
getContentManager().registerOntology(PeopleOntology.getInstance());
getContentManager().registerLanguage(new JCodec());
```

The `JCodec` uses Java serialization to encode and decode the content of messages; the `jade.content.lang.leap.LEAPCodec` provides CLDC-compliant encoding and decoding. The choice of the codec is not so relevant because the rest of the API is content-

language independent. The registration of ontologies and codecs is typically provided in the `setup()` method of the agent.

In order to send a message, we have two possibilities: through *concrete objects* or through *abstract descriptors*. The first approach is the easiest to use, but it is limited:

1) we create our content in terms of objects that belongs to the classes that we associated with schemas in the ontology, e.g., `Person` and `FatherOf` classes;
2) we use `fillContent()` in `ContentManager` to fill the content of the message.

The following code inform an agent that *"John lives in London and his only child Bill lives in Paris"*:

```
ACLMessage message = new ACLMessage(ACLMessage.INFORM);

// Set the fields of the ACL message
…

// Create the concrete object representing the content
Man john = new Man();
Man bill = new Man();
john.setName("John");
bill.setName("Bill");

Address johnAddress = new Address();
johnAddress.setCity("London");
john.setAddress(johnAddress);

Address billAddress = new Address();
billAddress.setCity("Paris");
bill.setAddress(billAddress);

FatherOf fatherOf = new FatherOf();
fatherOf.setFather(john);

List children = new ArrayList();
children.add(bill);

fatherOf.setChildren(children);

getContentManager().fillContent(message, fatherOf);
```

Using concrete objects like `john` and `fatherOf` is the easiest approach to filling the content of a message but it is not fully expressive. For examples, consider the following problem: we want to query an agent for the names of John's children. We need to send a query-ref message with the following content: `(iota ?X fatherOf(john, ?X))`, where `?X` is a variable that the receiver agent uses to come to know what we want to know. Such a content is an IRE, i.e., an

expression that identifies an object. The problem is that we cannot set the `children` attribute of a `FatherOf` object to a variable because such an attribute is a `List`. A number of techniques are available to solve this problem exploiting inheritance, but they all require that you implement many classes for describing the ontology. In order to solve this problem using only the classes we already implemented for the ontology, we introduced abstract descriptors. An abstract descriptor is an object that describes an instantiation of a schema, e.g., the following is the abstract descriptor that describes the concept "John":

```
AbsConcept absJohn = new AbsConcept(PeopleOntology.MAN);
absJohn.set(PeopleOntology.NAME, "John");
```

An abstract descriptor is created with the name of a class (`MAN` in this example). Then, we can set and get values on the descriptor using the names of the attributes. The structure of the descriptor, i.e., what the available attributes are and what are their values, must be coherent with the schema that the ontology associates with the name of the class (`MAN` in this example). The following is the code for performing the query about the names of John's children:

```
ACLMessage message = new ACLMessage(ACLMessage.QUERY_REF);

// Set the fields of the message
…

// Create the abstract descriptor representing the content
AbsConcept absJohn = new AbsConcept(PeopleOntology.MAN);
absJohn.set(PeopleOntology.NAME, "John");

AbsVariable  absX        = new AbsVariable("X")
AbsPredicate absFatherOf = new AbsPredicate(PeopleOntology.FATHER_OF);
absFatherOf.set(PeopleOntology.FATHER,   absJohn);
absFatherOf.set(PeopleOntology.CHILDREN, absX);

AbsIRE absIRE = new AbsIRE(absX, absFatherOf);

getContentManager().fillContent(message, absIRE);
```

The procedure for receiving a message is dual to that of sending a message. We can use both concrete objects and abstract descriptors, and if we try to create a concrete object from a message containing a variable, the content manager throws an `UngroundedException`. The following code handles inform messages:

```
ACLMessage msg = blockingReceive(ACLMessage.INFORM);

// The content of informs do not contain variables
Proposition p = (Proposition)getContentManager().extractContent(msg);
```

```
// Handle the content
if(p instanceof FatherOf) {
  …
}
```

If we want to handle incoming queries, we need to use `extractAbsContent()` to create an abstract descriptor from the message:

```
ACLMessage msg = blockingReceive(ACLMessage.QUERY_REF);

// The content of query-refs do contain variables
AbsIRE absIRE = (AbsIRE)getContentManager().extractAbsContent(msg);

// Handle the content
AbsVariable    absX = absIRE.getVariable();
AbsProposition absP = absIRE.getProposition();
```